

**4th Symposium on
Operating Systems
Design and
Implementation
(OSDI 2000)**

*San Diego, California, USA
October 23–25, 2000*

Sponsored by

The USENIX Association

Co-sponsored by **IEEE TCOS and ACM SIGOPS**

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
URL: <http://www.usenix.org>

The price is \$25 for members and \$32 for nonmembers.
Outside the U.S.A. and Canada, please add
\$12 per copy for postage (via air printed matter).

Past OSDI Proceedings

OSDI '99 (Third)	February 1999	New Orleans, Louisiana, USA	\$23/30
OSDI '96 (Second)	October 1996	Seattle, Washington, USA	\$20/27
OSDI '94 (First)	November 1994	Monterey, California, USA	\$20/27

© 2000 by The USENIX Association
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN 1-880446-16-2

Printed in the United States of America on 50% recycled paper, 10–15% post-consumer waste.

USENIX Association

**Proceedings of the
Fourth Symposium on Operating Systems
Design and Implementation
(OSDI 2000)**

Co-sponsored by IEEE TCOS and ACM SIGOPS

**October 23–25, 2000
San Diego, California, USA**

Conference Organizers

Program Co-Chairs

Michael B. Jones, *Microsoft Research*

Frans Kaashoek, *Massachusetts Institute of Technology*

Program Committee

Brian Bershad, *Appliant.com*

David Culler, *University of California at Berkeley*

Fred Douglass, *AT&T Labs—Research*

Peter Druschel, *Rice University*

David Johnson, *Carnegie Mellon University*

Butler Lampson, *Microsoft*

Andrew Myers, *Cornell University*

Dave Presotto, *Bell Laboratories*

Timothy Roscoe, *Sprint Labs*

Bill Weihl, *Akamai Technologies, Inc.*

John Wilkes, *HP Laboratories*

Steering Committee

Margo Seltzer, *Harvard University*

Karin Petersen, *Xerox PARC*

The USENIX Association Staff

External Reviewers

Atul Adya
David Anderson
Mohit Aron
Victor Bahl
Hari Balakrishnan
Gaurav Banga
Chuck Blake
Bill Bolosky
Josh Broch
Miguel Castro
John Chapin
Benjie Chen
Peter Chen
Albert M. K. Cheng
Alan Cox
Chuck Cranor
Douglass Decouto
John Douceur
Dan Duchamp
E. N. Elnozahy
Dawson Engler

Christof Fetzer
Marc Fiuczynski
Kevin Fu
Greg Ganger
R. Gopalakrishnan
Robert Grimm
John Hartman
John H. Hartman
Chris Hawblitzel
Y. Charlie Hu
Mark T.-i. Huang
Yennun Huang
Andrew Hume
Galen Hunt
Sitaram Iyer
John Jannotti
Michael Kaminsky
David Karger
Todd Knoblock
Eddie Kohler
Craig Labovitz

Jay Lepreau
Yinyang Li
Jochen Liedtke
David A. Maltz
Larry Masinter
Yaron Minsky
Jeff Mogul
Robert Morris
David Mosberger
Nathaniel Nystrom
Vivek Pai
Larry Peterson
Padmanabhan Pillai
Max Poletto
Misha Rabinovich
John Regehr
Rodrigo Rodrigues
Fred Schneider
Kang G. Shin
David Shur
Sandeep Sibal

Dan Simon
Fred Smith
Oliver Spatscheck
Peter Steenkiste
Bjarne Steensgaard
David Tarditi
Marvin Theimer
Jacobus Van der Merwe
Robbert van Renesse
Srinivasan Venkatachary
Phong Vo
Carl Waldspurger
Dan Wallach
Yi-Min Wang
Roger Wattenhofer
Emmett Witchel
Steve Zdancewic
Zheng Zhang
Lan Tian Zheng
Willy Zwaenepoel

4th Symposium on Operating Systems Design and Implementation

October 23–25, 2000
San Diego, California, USA

Index of Authors	vi
Message from the Program Chairs	vii

Monday, October 23

Applying Language Technology to Systems

Session Chair: David Culler, University of California at Berkeley

Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions	1
<i>Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem, Stanford University</i>	

Devil: An IDL for Hardware Programming	17
<i>Fabrice M��rillon, Laurent R��veill��re, Charles Consel, Renaud Marlet, and Gilles Muller, IRISA/INRIA</i>	

Taming the Memory Hogs: Using Compiler-Inserted Releases to Manage Physical Memory Intelligently	31
<i>Angela Demke Brown and Todd C. Mowry, Carnegie Mellon University</i>	

Scheduling

Session Chair: Timothy Roscoe, Sprint Labs

Surplus Fair Scheduling: A Proportional-Share CPU Scheduling Algorithm for Symmetric Multiprocessors	45
<i>Abhishek Chandra and Micah Adler, University of Massachusetts, Amherst; Pawan Goyal, Ensim Corp.; and Prashant Shenoy, University of Massachusetts, Amherst</i>	

Performance-Driven Processor Allocation	59
<i>Julita Corbal��n, Xavier Martorell, and Jes��s Labarta, Universitat Polit��cnica de Catalunya</i>	

Policies for Dynamic Clock Scheduling	73
<i>Dirk Grunwald and Philip Levis, University of Colorado; Keith I. Farkas, Compaq Computer Corp.; Charles B. Morrey III and Michael Neufeld, University of Colorado</i>	

Storage Management

Session Chair: Brian Bershad, Appliant.com

Towards Higher Disk Head Utilization: Extracting Free Bandwidth from Busy Disk Drives	87
<i>Christopher R. Lumb, Jiri Schindler, Gregory R. Ganger, and David F. Nagle, Carnegie Mellon University; Erik Riedel, Hewlett-Packard Labs</i>	

Latency Management in Storage Systems	103
<i>Rodney Van Meter, Quantum Corp., and Minxi Gao, University of California at Berkeley</i>	

A Low-Overhead, High-Performance Unified Buffer Management Scheme That Exploits Sequential and Looping References	119
<i>Jong Min Kim, Jongmoo Choi, and Jesung Kim, Seoul National University; Sam H. Noh, Hong-Ik University; Sang Lyul Min, Yookun Cho, and Chong Sang Kim, Seoul National University</i>	

Tuesday, October 24

Security

Session Chair: Butler Lampson, Microsoft Corp.

How to Build a Trusted Database System on Untrusted Storage135
Umesh Maheshwari, Radek Vingralek, and William Shapiro, InterTrust Technologies Corp.

End-to-End Authorization151
Jon Howell, Consystant Design Technologies; David Kotz, Dartmouth College

Self-Securing Storage: Protecting Data in Compromised Systems165
John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger, Carnegie Mellon University

Fast and Secure Distributed Read-Only File System181
Kevin Fu, M. Frans Kaashoek, and David Mazières, Massachusetts Institute of Technology

Networking

Session Chair: Peter Druschel, Rice University

Overcast: Reliable Multicasting with an Overlay Network197
John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and James W. O'Toole, Jr., Cisco Systems

System Support for Bandwidth Management and Content Adaptation in Internet Applications213
David Andersen, Deepak Bansal, and Dorothy Curtis, Massachusetts Institute of Technology; Srinivasan Seshan, Carnegie Mellon University; and Hari Balakrishnan, Massachusetts Institute of Technology

Storage Devices

Session Chair: John Wilkes, HP Laboratories

Operating System Management of MEMS-based Storage Devices227
John Linwood Griffin, Steven W. Schlosser, Gregory R. Ganger, and David F. Nagle, Carnegie Mellon University

Trading Capacity for Performance in a Disk Array243
Xiang Yu, Benjamin Gum, Yuqun Chen, Randolph Y. Wang, and Kai Li, Princeton University; Arvind Krishnamurthy, Yale University; and Thomas E. Anderson, University of Washington

Interposed Request Routing for Scalable Network Storage259
Darrell C. Anderson, Jeffrey S. Chase, and Amin M. Vahdat, Duke University

Wednesday, October 25

Reliability

Session Chair: David Johnson, Carnegie Mellon University

Proactive Recovery in a Byzantine-Fault-Tolerant System273
Miguel Castro and Barbara Liskov, Massachusetts Institute of Technology

Exploring Failure Transparency and the Limits of Generic Recovery289
David E. Lowell, Compaq Computer Corp.; and Subhachandra Chandra and Peter M. Chen, University of Michigan

Design and Evaluation of a Continuous Consistency Model for Replicated Services	305
<i>Haifeng Yu and Amin Vahdat, Duke University</i>	

System Architecture

Session Chair: Bill Weihl, Akamai Technologies, inc.

Scalable, Distributed Data Structures for Internet Service Construction	319
<i>Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler, University of California at Berkeley</i>	

Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java	333
<i>Godmar Back, Wilson H. Hsieh, and Jay Lepreau, University of Utah</i>	

Knit: Component Composition for Systems Software	347
<i>Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide, University of Utah</i>	

Index of Authors

Adler, Micah	45	Kim, Chong Sang	119
Andersen, David	213	Kim, Jesung	119
Anderson, Darrell C.	259	Kim, Jong Min	119
Anderson, Thomas E.	243	Kotz, David	151
Back, Godmar	333	Krishnamurthy, Arvind	243
Balakrishnan, Hari	213	Labarta, Jesús	59
Bansal, Deepak	213	Lepreau, Jay	333, 347
Brewer, Eric A.	319	Levis, Philip	73
Castro, Miguel	273	Li, Kai	243
Chandra, Abhishek	45	Liskov, Barbara	273
Chandra, Subhachandra	289	Lowell, David E.	289
Chase, Jeffrey S.	259	Lumb, Christopher R.	87
Chelf, Benjamin	1	Maheshwari, Umesh	135
Chen, Peter M.	289	Marlet, Renaud	17
Chen, Yuqun	243	Martorell, Xavier	59
Cho, Yookun	119	Mazières, David	181
Choi, Jongmoo	119	Mérillon, Fabrice	17
Chou, Andy	1	Meter, Rodney Van	103
Consel, Charles	17	Min, Sang Lyul	119
Corbalán, Julita	59	Morrey, Charles B., III	73
Culler, David	319	Mowry, Todd C.	31
Curtis, Dorothy	213	Muller, Gilles	17
Demke Brown, Angela	31	Nagle, David F.	87, 227
Eide, Eric	347	Neufeld, Michael	73
Engler, Dawson	1	Noh, Sam H.	119
Farkas, Keith I.	73	O'Toole, James W., Jr.	197
Flatt, Matthew	347	Reid, Alastair	347
Fu, Kevin	181	Réveillère, Laurent	17
Ganger, Gregory R.	87, 165, 227	Riedel, Erik	87
Gao, Minxi	103	Scheinholtz, Michael L.	165
Gifford, David K.	197	Schindler, Jiri	87
Goodson, Garth R.	165	Schlosser, Steven W.	227
Goyal, Pawan	45	Seshan, Srinivasan	213
Gribble, Steven D.	319	Shapiro, William	135
Griffin, John Linwood	227	Shenoy, Prashant	45
Grunwald, Dirk	73	Soules, Craig A. N.	165
Gum, Benjamin	243	Stoller, Leigh	347
Hallem, Seth	1	Strunk, John D.	165
Hellerstein, Joseph M.	319	Vahdat, Amin M.	259, 305
Howell, Jon	151	Vingralek, Radek	135
Hsieh, Wilson H.	333	Wang, Randolph Y.	243
Jannotti, John	197	Yu, Haifeng	305
Johnson, Kirk L.	197	Yu, Xiang	243
Kaashoek, M. Frans	181, 197		

Message from the Symposium Chairs

OSDI 2000 captures exciting, innovative work in the systems software area, taking a broad view of what the area encompasses—work we believe that you, as a systems practitioner, will find both interesting and useful. The program presents important results in a wide range of areas including new storage devices, fault tolerance, software architectures for Internet services, Internet content distribution, applying language techniques to systems software, scheduling for variable-speed processors, storage management, and distributed secure filesystems.

It was our goal for OSDI 2000 to have a program of creative, highly original work opening up new areas—not merely incremental results building on prior work. We were able to achieve this goal even beyond our highest expectations due to the diverse collection of innovative work submitted by the authors. We accepted 24 out of the 111 submissions and received far more great papers than we were able to accommodate in a 2 1/2-day program. We believe that you'll agree that the conference offers an exceptionally strong and diverse program.

The program is the result of the hard work of many. First, we want to thank the authors who submitted papers, including those whose papers we weren't able to accept. Next, we want to thank the program committee members for their diligent labor. They read a large number of papers in a short time, producing 182 pages of reviews, and shepherded the accepted papers. We also want to thank the external reviewers, who contributed numerous high-quality reviews. Finally, we would like to thank the USENIX staff for all of their detailed and timely help in putting the conference together.

Our thanks to each of these individuals for their contributions towards producing the outstanding OSDI 2000 program that you see before you. It's been our privilege working with each of you.

Sincerely,

Michael B. Jones, *Microsoft Research*

Frans Kaashoek, *Massachusetts Institute of Technology*

Symposium Co-Chairs

Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions

Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem*

Computer Systems Laboratory

Stanford University

Stanford, CA 94305, U.S.A.

Abstract

Systems software such as OS kernels, embedded systems, and libraries must obey many rules for both correctness and performance. Common examples include “accesses to variable A must be guarded by lock B,” “system calls must check user pointers for validity before using them,” and “message handlers should free their buffers as quickly as possible to allow greater parallelism.” Unfortunately, adherence to these rules is largely unchecked.

This paper attacks this problem by showing how system implementors can use *meta-level compilation* (MC) to write simple, system-specific compiler extensions that automatically check their code for rule violations. By melding domain-specific knowledge with the automatic machinery of compilers, MC brings the benefits of language-level checking and optimizing to the higher, “meta” level of the systems implemented in these languages. This paper demonstrates the effectiveness of the MC approach by applying it to four complex, real systems: Linux, OpenBSD, the Xok exokernel, and the FLASH machine’s embedded software. MC extensions found roughly 500 errors in these systems and led to numerous kernel patches. Most extensions were less than a hundred lines of code and written by implementors who had a limited understanding of the systems checked.

1 Introduction

Systems software must obey many rules such as “check user permissions before modifying kernel data structures,” “for speed, enforce mutual exclusion with spin locks rather than disabling interrupts,” and “message handlers must free their buffer before completing.”

*This research was supported in part by DARPA contract MDA904-98-C-A933 and by a Terman Fellowship.

Code that does not obey these rules can degrade performance or crash the system.

There are several methods to find violations of system rules. A rigorous way is to build an abstract specification of the code and then use model checkers [23, 32] or theorem provers/checkers [2, 11, 25] to check that the specification is internally consistent. When applicable, formal verification finds errors that are difficult to detect by other means. However, specifications are difficult and costly to construct. Further, specifications do not necessarily mirror the code they abstract and, in practice, suffer from missing features and over-simplifications. While recent work has begun attacking these problems [6, 14], it is extremely rare for software to be verified.

The most common method used to detect rule violations is testing. Testing is simpler than verification. It also avoids the mirroring problems of formal verification by working with actual code rather than an abstraction of it. However, testing is dynamic, which has numerous disadvantages. First, the number of execution paths typically grows exponentially with code size. Thorough, precise testing requires writing many test cases to exercise these paths and drive the system into error states. The effort required to create these tests, and the time it takes to run them, scales with the amount of code. As a result, real systems have many paths that are rarely or never hit by testing and errors that manifest themselves only after days of continuous execution. Further, finding the cause of a test failure can be difficult, especially when the effect is a delayed system crash. Finally, testing requires running the tested code, which can create significant practical problems. For example, testing all device drivers in an OS requires acquiring possibly hundreds or thousands of devices and understanding how to thoroughly exercise them.

Another common method to detect rule violations is manual inspection. This method has the strength that it can consider all semantic levels and adapt to ad hoc coding conventions and system rules. Unfortunately, many systems have millions of lines of code

with deep, complex code paths. Reasoning about a single path can take minutes or sometimes, when dealing with concurrency, hours. Further, the reliability of manual inspection is erratic.

These methods leave implementors in an unfortunate situation. Verification is impractical for most systems. Testing misses many cases and makes diagnosis difficult. Manual inspection is unreliable and tedious. One possible alternative is to use static compiler analysis to find rule violations. Unlike verification, compilers work with the code itself, removing the need to write and maintain a specification. Unlike testing, static analysis can examine all execution paths for errors, even in code that cannot be conveniently executed. Further, a compiler analysis pass reduces the need to construct numerous test cases and scales from a single function to an entire system with little increase in manual effort.

Compilers can be used to enforce systems rules because many rules have a straightforward mapping to program source. Rule violations can be found by checking when source operations do not make sense at an abstract level. For example, ordering rules such as “interrupts must be enabled after being disabled” reduce to observing the order of function calls or idiomatic sequences of statements (in this case, a call to a disable interrupt function must be followed by a re-enable call).

The main barrier to a compiler checking or optimizing at this level is that while it must have a precise understanding of the semantics of its input code, it typically has no idea of the “meta” semantics of the software system this code constructs. Thus, it cannot check many properties inexpressible (or just not expressed) in terms of the underlying language’s type system. This leaves an unfortunate dichotomy. Implementors understand the semantics of the system operations they build and use but do not have the mechanisms to check or exploit these semantics automatically. Compilers have the machinery to do so, but their domain ignorance prevents them from exploiting it.

This paper shows how to automatically check systems rules using *meta-level compilation* (MC). MC attacks this problem by making it easy for implementors to extend compilers with lightweight, system-specific checkers and optimizers. Because these extensions can be written by system implementors themselves, they can take into account the ad hoc (sometimes bizarre) semantics of a system. Because they are compiler based, they also get the benefits of automatic static analysis.

In our MC system, implementors write extensions in a high-level state-machine language, *metal*. These

extensions are dynamically linked into our extensible compiler, *xg++*, and applied down all flow paths in all functions in the program source input. They use language-based patterns to recognize operations that they care about. Then, when the input code matches these patterns, they detect rule violations by transitioning between states that allow or disallow other operations.

This paper’s primary contribution is its demonstration that MC is a general, effective approach for finding system errors. Our most important results are:

1. MC checkers find serious errors in complex, real systems code. We present a series of extensions that found roughly 500 errors in four systems: the Linux 2.3.99 kernel, OpenBSD, the Xok exokernel [16], and the FLASH machine’s embedded cache controller code [20]. Many errors were the worst type of systems bugs: those that crash the system, but only after it has been running continuously for days.
2. MC optimizers discover system-level opportunities that are difficult to find with manual inspection. While the main focus of this paper is error checking, MC extensions can also be used for optimization. Section 8 describes three FLASH-specific, MC optimizers that found hundreds of system-level optimization opportunities.
3. MC extensions are simple. The extensions mentioned above are typically less than a hundred lines of code.

A practical result of our experience with MC is that the majority of our extensions were written by programmers who had only a passing familiarity with the systems that they checked. Although writing code that obeys system rules can be quite difficult, these rules are easy to express. Thus, writing checkers for many of them is relatively straightforward.

This paper is laid out as follows. Section 2 discusses related work. Section 3 gives an overview of MC and the system we use to implement it. Section 4 applies the approach to the C `assert` macro and shows that even in such a limited domain, MC provides non-trivial benefits. Section 5 shows how to use MC to enforce ordering constraints such as checking that kernels verify user pointers before using them. Section 6 extends this to global, system-wide constraints. Section 7 is a more detailed case study in how we used MC to check Linux locking and interrupt disabling/re-enabling disciplines. Section 8 describes our FLASH optimizers, and Section 9 concludes.

2 Related Work

We proposed the initial idea of MC in [9] and provided a simple system, *magik* (based on the *lcc* ANSI C compiler [12]), for using it. While the original paper had many examples, it provided no experimental evaluation. This paper provides a more developed view of MC, a significantly easier-to-use and more powerful framework for building extensions, and an experimental demonstration of its effectiveness. Concurrently with this paper, we presented a detailed case study of applying MC to the FLASH system [4]. The 8 compiler extensions presented in that paper discovered 34 errors in FLASH code that could potentially crash the machine, such as message handlers that lost or double freed hardware message buffers and buffer race conditions. This paper's main difference is its demonstration that MC is a general technique by applying it to a variety of systems. Because of this broader scope, it lacks the detail in [4], but finds roughly a factor of ten more errors.

Below, we compare our work to efforts in high-level compilation, verification, and extensible compilers.

Higher-level compilation. Many projects have hard wired application-level information in compilers. These projects include: compiler-directed management of I/O [24]; the ERASER dynamic race detection checker [30]; ParaSoft's Insure++ [19], which can check for Unix system call errors; the use of static analysis to check for security errors in privileged programs [1]; and the GNU compilers' `-Wall` option, which warns about dangerous functions and questionable programming practices. Related to the checkers in this paper, Microsoft has an internal tool for finding a fixed set of coding violations in Windows device drivers [27] such as errors in handling 64-bit code and missing user pointer validity checks.

These projects use compiler support to analyze specific problems, whereas MC explicitly argues for the general use of compilers to check and optimize systems and provides an extensible framework for doing so. This extensibility enables detection of rule violations that are impossible to find without system-specific knowledge.

Systems for finding software errors. Most approaches to statically finding software errors center around either formal verification (as discussed in Section 1) or strong type checking.

Verification uses stronger analysis than MC extensions. However, MC extensions appear to be more generally effective. To the best of our knowledge, verification papers tend to find a small number of errors (typically 0-2), whereas the MC checkers in this paper

found hundreds. Verification's lower bug counts seem largely due to the difficulty in writing specifications, which scales with code size. As a consequence, only small pieces of code are verified. In contrast, because MC operates directly on source code, it (like traditional compiler analyses) applies as easily to millions of lines of code as it does to only a few.

Two recent strong-typing systems are the extended static type checking (ESC) project [8] and Intrinsa's PREFIX [15]. Both of these systems use stronger analyses than our approach. However, they only check for a fixed set of low-level errors (e.g., buffer overruns and null pointer references). Their lack of extensibility means that, with the exception of ESC's support for finding some class of race conditions, neither system can find the system-level errors that MC can detect.

LCLint [10] statically checks programmer source annotations to detect coding errors and abstraction barrier violations. Like ESC and Intrinsa, LCLint is not extensible, which prevents it from finding the errors that MC can find. Further, the source annotations that LCLint requires scale with code size, significantly increasing the manual effort needed to apply it.

Extensible compilation. There have been a number of "open compiler" systems that allow programmers to add analysis routines, usually modeled as extensions, that traverse the compiler's abstract syntax tree. These include Lord's *ctool* [22], which allows scheme extensions to walk over an abstract syntax tree for C, and Crew's Prolog-based AST-LOG [7], also used for C.

Lamping et al. [21] and Kiczales et al. [17] argue for pushing domain-specific information into compilation. They use meta-object protocols (MOPs) to allow programs to be augmented with a "meta" part that controls the base [17]. Such protocols are typically dynamic and have fairly limited analysis abilities. Shigeru Chiba's *Open C++* [3] provides a static MOP that allows users to extend the compilation process.

The extensions in these systems are mainly limited to syntax-based tree traversal or transformation and do not have data flow information. As a result, they seem to be both less powerful than MC extensions and more difficult to use. Our current, language-based approach is a dramatic improvement over our previous tree-based systems: extensions are 2-4 times smaller, have less bugs, and handle more cases. Further, to the best of our knowledge, *ctool*, *ASTLOG*, and *Open C++* provide no experimental results, making it difficult to evaluate their effectiveness.

At a lower-level, the ATOM object code modifi-

cation system [31] gives users the ability to modify object code in a clean, simple manner. By focusing on machine code, ATOM can be used in more situations than MC, which requires source code. However, while dynamic testing schemes [13, 30] are well served by object-level modifications, it would be difficult to perform our static checks without the semantic information available in the compiler.

Concurrently with our original work [9], Kiczales et al. [18] proposed “aspect oriented programming” (AOP) as a way of combining code that manages “aspects,” such as synchronization, with code that needs them. AOP has the advantage of being integrated within a traditional language framework. It has the disadvantage that aspects have more limited scope than MC extensions, which survey the entire system as well as check rules difficult to enforce with an AOP framework (e.g., preventing kernel code from using floating point). Further, because AOP requires source modifications, retro-fitting it on the systems we check would be non-trivial.

3 Meta-level Compilation

Many systems constraints describe legal orderings of operations or specific contexts in which these operations can or cannot occur. Since the actions relevant to these rules are visible in program source, an MC compiler extension can check them by searching for the corresponding operations and verifying that they obey the given ordering and/or contextual restrictions. Table 1 gives a representative set of rule “templates” that can be checked in this manner along with examples. Many system rules that roughly follow these templates can be checked automatically. For example, an MC extension to enforce the contextual rule, “for speed, if a shared variable is not modified, protect it with read locks,” can search for each write-lock critical section, examine all variable uses, and, if no stores occur to protected variables, demote the locks or suggest alternative usage.

3.1 Language Overview

In our implementation of MC, compiler extensions are written in a high-level, state-machine language, *metal* [5]. These extensions are dynamically linked into our extensible compiler, *xg++* (based on the GNU *g++* compiler). After *xg++* translates each input function into its internal representation, the extensions are applied down every possible execution path in that function. The state machine part of the language can be viewed as syntactically similar to a “yacc” specification. Typically, SMs use patterns

```
{ #include "linux-includes.h" }
sm check_interrupts {
  // Variables
  // used in patterns
  decl { unsigned } flags;

  // Patterns
  // to specify enable/disable functions.
  pat enable = { sti(); }
               | { restore_flags(flags); } ;
  pat disable = { cli(); };

  // States
  // The first state is the initial state.
  is_enabled: disable ==> is_disabled
    | enable ==> { err("double enable"); }
    ;
  is_disabled: enable ==> is_enabled
    | disable ==> { err("double disable"); }
    // Special pattern that matches when the SM
    // hits the end of any path in this state.
    | $end_of_path$ ==>
      { err("exiting w/intr disabled!"); }
    ;
}
```

Figure 1: A *metal* SM to detect (1) when interrupts disabled using `cli` are not re-enabled using either `sti` or `restore_flags` and (2) duplicate enable/disable calls.

to search for interesting source code features, which, when matched, cause transitions between states. Patterns are written in an extended version of the base language (C++), and can match almost arbitrary language constructs such as declarations, expressions, and statements. Expressing patterns in the base language makes them both flexible and easy to use, since they closely mirror the source constructs they describe.

Figure 1 shows a stripped-down *metal* extension for Linux that checks that disabled interrupts are re-enabled or restored to their initial state upon exiting a function. Interrupts are disabled by calling the `cli()` procedure; they are enabled by calling `sti()` or restored using `restore_flags(flags)`, where the `flags` variable holds the interrupt state before the `cli()` was issued. Conceptually, the extension finds violations by checking that each call to disable interrupts has a matching enable call on all outgoing paths. As refinements, the extension warns of duplicate calls to these functions or non-sequitur calls (e.g., re-enabling without disabling). A more complete version of this checker, described in Section 7, found 82 errors in Linux code.

The extension tracks the interrupt status using

Rule template	Examples
"Never/always do X"	"Do not use floating point in the kernel." (§ 4.3) "Do not allocate large variables on the 6K byte kernel stack." (§ 4.3) "Do not send more than two messages per virtual network lane." "Allocate as much storage as an object needs." (§ 5.2)
"Do X rather than Y"	"Use memory mapped I/O rather than copying." "Avoid globally disabling interrupts."
"Always do X before/after Y"	"Check user pointers before using them in the kernel." (§ 5.1) "Handle operations that can fail (e.g., memory, disk block, virtual interrupt allocation)." (§ 5.2) "Re-enable interrupts after disabling them." (§ 7) "Release locks after acquiring them." (§ 7) "Check user permissions before modifying kernel data structures."
"Never do X before/after Y"	"Do not acquire lock A before B." "Do not use memory that has been freed." (§ 5.2) "Do not (deallocate an object, acquire/release a lock) twice." (§ 5.2 § 7) "Do not increment a module's reference count after calling a function that can sleep." (§ 6.3)
"In situation X, do (not do) Y"	"Protect all variable mutations with write locks." "If a system call fails, reverse all side-effect operations (deallocate memory, disk blocks, pages, unincrement reference counters)." (§ 5.2 § 6.3) "To avoid deadlock, while interrupts are disabled, do not call functions that can sleep." (§ 6.2)
"In situation X, do Y rather than Z"	"If a variable is not modified, protect it with read locks." "If code does not share data with interrupt handlers, then use spin locks rather than the more expensive interrupt disabling." "To save an instruction when setting a message opcode, xor in the new and old opcode rather than using assignment." (§ 8)

Table 1: Sample system rule templates and examples. Checkers for the rule are denoted by section number.

```

/* From Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head *sh,
                int b_size) {
    struct buffer_head *bh;
    unsigned long flags;

    save_flags(flags);
    cli();
    if ((bh = sh->buffer_pool) == NULL)
        return NULL;
    sh->buffer_pool = bh->b_next;
    bh->b_size = b_size;
    restore_flags(flags);
    return bh;
}

```

Figure 2: Example code from the Linux 2.3.99 Raid 5 driver illustrating a real error caught by the extension. The SM will be applied down both paths in this function. The path ending with a return of `bh` is well formed and will be accepted. The path ending with the return of `NULL` is not, and will get a warning about not re-enabling interrupts.

two states, `is_enabled` and `is_disabled`. SMs start in the state mentioned in the first transition definition (here, `is_enabled`). Each state has a set of rules specifying a pattern, an optional state transition, and an optional action. Actions can be arbitrary C++ code. For a given state, *metal* checks pattern rules in lexical order. If any code matches the specified patterns, *metal* processes this matching code, sets the state to the new state (the token after the `==>` operator), and executes the action. In this example, `is_enabled` has two rules. The first, actionless rule searches for functions that disable interrupts using the `disable` pattern and transitions to the `is_disabled` state. The second rule searches for calls to functions that enable interrupts and gives a warning. Since it does not specify a transition state, the SM remains in the `is_enabled` state. If no pattern matches, the SM remains in the same state and continues down the current code path. The `flags` variable is a wild card that matches any expression of type `unsigned`. When it is matched, *metal* will put the matching expression in `flag`, which can then be used in an action. We use this feature in an extension discussed in Section 4.

To run this SM, it is first compiled with `mcc`, our *metal* compiler. It is then dynamically linked into `xg++` using a compile-time, command-line flag. When run on the Linux "RAID 5" driver buffer allocation code in Figure 2, it is pushed down both paths

in the function. The first path returns NULL when the buffer pool is empty (i.e., when the `if` statement fails); the other returns a buffer on successful allocation. The first path fails to re-enable interrupts, and this error¹ is caught and reported by the extension.

One way to get a feel for how costly it would be to manually perform the check our SM does automatically is that even when we showed an experienced Linux programmer the exact error in Figure 2, it took him over 20 minutes to examine a single call chain out of the nine leading to this function. Performing similar analysis for the other hundreds of thousands of lines of driver and kernel code seems impractical.

3.2 Practical issues

Metal SMs can specify whether they should be applied either down all paths (i.e., flow-sensitive) or linearly through the code (i.e., flow-insensitive). A simple implementation of flow-sensitive SMs could take exponential time in some cases. We use aggressive caching to prune redundant code paths where SM instances follow paths that join (e.g., `if` statements, loops) and reach the join point in the same state. Our caching is based on the fact that a deterministic SM applied to the same input in the same internal state must compute the same result. The system represents the state of an SM as a vector holding the value of its variables. For each node in the input flow-graph, it records the set of states in which it has been visited. If an SM arrives at a node in the same state as a previous instance, the system prunes it.

While caching was originally motivated by speed, perhaps its most important feature is that it provides a clean framework for computing loop “fixed points” transparently. When an SM has exhausted the set of states reachable within the loop (typically with two iterations), *metal* automatically stops traversing the loop. This fixed-point behavior depends on the SM having a finite (and small) number of states. We do not currently enforce this restriction.

The current *xg++* system does not integrate global analysis with the SM framework. Instead, it provides a library of routines to emit client-annotated flow graphs to a file, which can then be read and traversed. Section 6 gives an example of how we used this framework to compute the transitive closure of all possibly-sleeping functions. We are integrating these two passes.

¹Amusingly, this interrupt disable bug would be masked by an immediate kernel segmentation fault since callers of this function dereference the returned pointer without checking whether the allocation succeeded.

3.3 Caveats

Most of our extensions are checkers rather than verifiers: they find bugs, but do not guarantee their absence. For example, their ignorance of aliases prevents them from asserting that many actions “cannot happen.” In general, many compiler problems are undecidable, which places hard limits on the effectiveness of static analysis. Despite these limitations, as our results show, MC extensions are quite effective. We are currently investigating how to turn some classes of checkers into verifiers.

We mainly check systems we did not build. As a result, some rule violations we found might not be bugs because the code could use a non-obvious system feature that works correctly in a specific situation. We countered this danger in two ways. First, we sent our error logs to the system implementors of Linux, FLASH, and Xok for confirmation. However, while we got feedback on many errors, their sheer number meant that many did not receive careful examination. Second, we conservatively did not count many cases that were difficult to reason about. While our results may still contain mis-diagnoses, we would be surprised if these caused more than a few percentage points difference.

Several of our checkers produce a number of false positives (in the worst case, in Section 7, up to three per error). These are due to the limitations of both static analysis and our checkers, which primarily use simple local analyses. Usually these numbers can be reduced significantly by adding some amount of global analysis or system-specific knowledge. In almost all cases, each false positive can be suppressed with a single source annotation. Extensions can provide annotations by supplying a set of reserved functions that clients call to indicate that a specific source-level warning should be suppressed. As a refinement, checkers can detect bogus or erroneous annotations by warning when they are not needed.

Basing our MC system on a C++ compiler has caused difficulties when applying it to Linux and Xok. These systems aggressively assume C’s more relaxed type system and use GNU extensions that are illegal in g++. Thus, while in theory MC can be applied to a system transparently, we had to modify Xok and Linux to remove GNU C constructs that are illegal in C++. We also modified the g++ front-end to relax its type checking. To avoid this labor for other systems, we are currently finishing a gcc-based implementation of *xg++*. More generally, since the *metal* language has been designed to be shielded from both the underlying language and compiler, we plan to port it to other languages and other compilers.

The remainder of this paper describes the extensions we implemented using *metal* and *xg++* and the results of applying the concept of meta-level compilation to real systems.

4 A Simple Meta-language

The C `assert` macro takes a single condition as its argument, checks this condition at runtime, and aborts execution if the condition is false. This macro defines one of the simplest meta-languages possible: it has no state and a single operation. This section shows how MC can help even such simple interfaces by presenting two extensions that check the following two assertion invariants:

1. Assertions should not have non-debugging side-effects. Frequently, `assert` is used only for development and turned off in production code. If an `assert` condition has important side-effects, these will disappear and the program will behave incorrectly.
2. Assertion conditions should not fail. Programmers use assertions to check for conditions that should not happen. Any code path leading to an assertion that causes its boolean expression to fail is probably a bug.

4.1 Checking assertion side-effects

Figure 3 presents a *metal* checker that inspects assertion expressions for side-effects. The directive, “`flow.insensitive`,” tells *metal* to apply the extension linearly over input functions rather than down all paths, improving speed and error reporting (since there will be exactly one message per violation). The SM begins in the initial state, `start`, and uses the literal *metal* pattern “`{assert(expr);}`” to find all `assert` uses.² On each match, *metal* stores the `assert` expression in the variable, `expr`. It then runs `start`’s action, which uses the *metal* procedure `mgk_expr_recurse` to recursively apply the SM to the expression in `expr` in the `in_assert` state. The `in_assert` state uses *metal*’s generic type “`any`” to match assignments, and pointer increments and decrements of any type. Note that the assignment operator will also detect uses of C’s infix operators (e.g., `+=`, `-=`, etc.). The extension matches any function call with any set of arguments using the extended types

²Since patterns can match nearly arbitrary C code, it does not matter if `assert` is a function or a macro; we have modified the pre-processor to ignore line and file directives.

```
{ #include <assert.h> }
// Apply SM ignoring control flow
sm Assert flow_insensitive {
  // Match expressions of "any" type
  decl { any } expr, x, y, z;
  // Used in combination to match all
  // calls with any arguments
  decl { any_call } any_fcall;
  decl { any_args } args;

  // Find all assert calls. Then apply
  // SM to "expr" in state "in_assert."
  start: { assert(expr); } ==>
    { mgk_expr_recurse(expr, in_assert); } ;
  // Find all side-effects
  in_assert:
    // Match all calls
    { any_fcall(args) } ==>
      { err("function call"); }
    // Match any assignment (including
    // the operators +=, -=, etc.)
    | { x = y } ==> { err("assignment"); }
    // Match all increments and decrements
    // --z and ++z omitted for brevity
    | { z++ } ==> { err("post-increment"); }
    | { z-- } ==> { err("post-decrement"); } ;
}
```

Figure 3: A *metal* SM that warns of side-effects in `assert` uses.

`any_call` and `any_args` in combination. To assist developers in writing extensions, *metal* provides a set of generic types for matching different classes of types (e.g., scalars, pointers, floats), and different programming constructs (e.g., case labels, indirections).

When applied to Xok’s ExOS library operating system, this 25 line extension found 16 violations in 199 `assert` uses. Two were false positives triggered by debugging functions. These could be suppressed by wrapping such calls in a differently named, unchecked assertion macro. The remaining fourteen cases were errors in crucial system code that would function incorrectly if the assertion was removed. The underlying cause of these errors was `assert`’s use as shorthand for checking the result of possibly-failing operations such as insertion of page table entries and deallocation of shared memory regions. A typical example is the following snippet from the ExOS “`mmap`” code to insert a page table entry:

```
/* libexos/os/mmap.c:mmap_fault_handler:410 */
assert(_exos_self_insert_pte(0, PG_P|
  PG_U|PG_W, PGROUNDDOWN(va), 0, NULL) == 0);
```

The effect of removing the `assert` condition (and hence these calls) would be mysterious virtual memory errors.

4.2 Checking assertions statically

Assertions specify conditions that the programmer believes must hold. Without MC, compilers are oblivious to this fact, so `assert` checks can only occur dynamically. With MC, it is possible to find errors by evaluating these conditions statically, thereby quickly and precisely finding errors.

We wrote such an extension on top of *xg++*. At a high level, it uses *xg++*'s dataflow routines to track the values of scalar variables. At each `assert` use, it evaluates the assertion expression against these known set of values. If the expression could fail, it emits a warning. Currently, *xg++* only performs primitive analysis that tracks the set of constant assignments to scalar variables on a given path. The set of possible values for a variable is then just the union of constant assignments to that variable before it is used. If any non-constant assignments occur, the value is considered "unknown." Returning the set of possible values allows the effectiveness of the checker to transparently increase as our analysis in *xg++* becomes more powerful. As a practical refinement, we eliminate a large class of false positives by ignoring assertions of the constant "0" (which always fails) since this is an idiomatic method for programmers to terminate execution in "impossible" situations.

When applied to the FLASH cache coherence code (discussed more in Section 8) the 100 line extension found five errors that could have crashed the system. These errors underscore the value of static evaluation, since they were in code that had been heavily tested for over five years. They had been missed because the length and complexity of typical FLASH code paths caused them to only occur sporadically. This complexity also makes manual detection of errors difficult. On one path, the assignment and the assertion that it violated were 300 lines apart and separated by 20 if statements, 6 else clauses, and 10 conditional compilation directives. Another case beat this by having 21 if statements, 4 else clauses, and 29 conditional compilations! Even given the exact situation that leads to the error, inspecting such paths is mind-numbing.

4.3 Discussion

Library implementations cannot inspect the context in which they are used or how they are invoked. MC can be used to attack these blindnesses. Our first extension used MC to detect illegal actions in `assert` uses, something that an `assert` implementation cannot otherwise do either dynamically or statically. Our second extension used context knowledge to push dynamically evaluated conditions to compile time. A

similar approach can be used to make certain dynamic error checks static or to improve performance by allowing implementations to specialize themselves to a given context, such as a memory allocator that generates specialized inline allocations for constant size allocation requests.

The restriction on side-effects in assertion conditions is a miniature example of a more general pattern of "language subsetting," where systems impose an execution context more restrictive than the base language in which code is written. We have built two other extensions that enforce system-specific execution restrictions. The first warns when kernel code uses floating point. It found one case where a Linux graphics driver assumes that floating point calculations will be evaluated at compile time. Using a compiler other than gcc or lower optimization levels could violate this assumption. The second checks for stack overflow. It found 10 places where Linux code allocated variables larger than 3K on the 6K kernel stack, and numerous 1K or larger allocations. Most of these led to patches by kernel maintainers. It also found a similar case in Xok where an innocent looking stack-allocated structure turned out to be over 8K bytes.

In addition to checking, systems can use restriction checkers for optimization by detecting when an application's actions are more limited than the general case. For example, a threads package can use smaller stack sizes than the default if it can derive an upper bound on stack usage.

5 Temporal Orderings

Many system operations must (or must not) happen in sequence. Sequencing rules are well-suited for compiler checking since sequences are frequently encoded as literal procedure calls in code. This allows a *metal* extension to find violations by searching for operations and transitioning to states that allow, disallow, or require other operations. This section discusses two such extensions. The first enforces an "X before Y" rule that system calls properly check application pointers passed to them for validity before using them. The second checks that code obeys a set of ordering rules for memory allocation and deallocation.

5.1 Checking copyin/copyout

Most operating systems guard against application corruption of kernel memory by, in part, using special routines to check system call input pointers and to move data between user and kernel space. We present an MC extension that finds errors in such code by

finding paths where an application pointer is used before passing through such routines. At each system call definition, the extension uses a special *metal* pattern to find every pointer parameter, which it binds to a tainted state. (The use of per-variable state differs from the previous checkers that used a single, global state per path.) The only legal operations on a tainted variable are being (1) killed by an assignment or (2) passed as an argument to functions expecting tainted inputs (e.g, data movement routines or output functions such as `kprintf`). All other uses will be signaled as an error.

We tailored a version of this checker for the Xok exokernel code. It detects which procedures are system calls using the exokernel naming convention that such routine names begin with the prefix “`sys_`.” As a refinement, the checker warns when any non-system-call routines use “paranoid” user-data routines. It examined 187 distinct user pointers in the exokernel proper and device code and found 18 errors. A typical error is this command to issue disk requests:

```
/* from sys/kern/disk.c */
int sys_disk_request (u_int sn, struct Xn_name
    *xn_user, struct buf *reqbp, u_int k) {
    ...
    /* bypass for direct scsi commands */
    if (reqbp->b_flags & B_SCSCMD)
        return sys_disk_scsicmd (sn, k, reqbp);
```

Here, the pointer, `reqbp`, is passed in from user space and dereferenced in the `if` statement without being checked.

This extension also signalled 15 false positives. Four of these were due to a stylized use where non-null pointers were verified using standard routines, but null ones were allowed through (they would be handled correctly by lower levels). Three others were due to kernel backdoors used to let system calls call other system calls with unchecked parameters. The remaining were due to the checker’s lack of global analysis and its disallowing of tainted variable copies.

5.2 Checking memory management

Most kernel code uses memory managers based loosely on the C procedures `malloc` and `free`. We present an extension that checks four common rules:

1. Since memory allocation can fail, kernel code must check whether the returned pointer is valid (i.e., not null) before using it.
2. Memory cannot be used after it has been freed.
3. Paths that allocate memory and then abort with an error should typically deallocate this memory before returning.

Violation	Linux		OpenBSD	
	Bug	False	Bug	False
No check	79	9	49	2
Error leak	44	49	3	1
Use after Free	7	3	0	0
Underflow	2	0	0	0
Total	132	61	52	3

Table 2: Error counts for Linux and OpenBSD. The checker was applied 4268 times in Linux and 464 times in OpenBSD.

4. The size of allocated memory cannot be less than the size of the object the assigned pointer holds.

Figure 4 shows a stripped-down extension that checks these rules. For space, the size check and most error reporting code is omitted. This extension, like the previous one, associates each variable with a state encoding what operations are legal on it. Pointers to allocated storage can be in exactly one of four states: `unknown`, `null`, `not_null`, or `freed`. A variable is bound to the `unknown` state at every allocation site. When an `unknown` variable is compared to `null` (e.g., in C, “0”) the extension sets the variable’s state on the true (`null`) path to `null` and on the false (`non-null`) path to `not_null`. When the variable is compared to `non-null`, these two cases are reversed. The two initial patterns recognize C’s check-and-compare allocation idiom and combine these transitions with the initial variable binding. Pointers passed to `free` transition to the `freed` state. As a minor refinement, when variables are overwritten, the extension stops following them by transitioning to the special *metal* state, `stop`.

The checker only allows dereferences of `not_null` pointers. This restriction catches instances when memory is used before being checked, on `null` paths, or after being freed. It catches double-free errors by warning when `freed` pointers are passed to `free`. It catches cases when error paths do not free allocated memory by warning when any `non_null` or `unchecked` variable reaches a return of a negative integer, which idiomatically signals an error path.

The full version of the checker is 60 lines of code. We get a lot for so little: the extension implements a flow-sensitive compiler analysis pass that checks for rules on all paths and takes into consideration the observations furnished by passing through conditionals. As Table 2 shows, the extension found 132 errors in Linux and 51 errors in OpenBSD. It turned up 61 and 3 false positives respectively, most due to not

handling variable copies, or not detecting when allocated memory would be freed by a cleanup routine.

The most common error was not checking the result of memory allocation: 79 cases in Linux, 49 in OpenBSD. In Linux, the single largest source of these errors was an allocation macro, `CODA_ALLOC`, which was widely used throughout the Coda file system code. It contains the unfortunate code:

```
/* include/linux/coda_linux.h:CODA_ALLOC */
ptr = (cast)vmalloc((unsigned long) size);
...
if (ptr == 0)
    printk("kernel malloc returns 0 at %s:%d\n",
        __FILE__, __LINE__);
memset( ptr, 0, size );
```

While this code prints a helpful message on every failed allocation, the initialization using `memset` will immediately cause a kernel segmentation fault.

The next most common error was not freeing memory on error paths (44 in Linux, 3 in OpenBSD). A typical not-freing error is given in Figure 5. An idiomatic mistake was to have many exit points from a function, but forgetting to free the memory at all of these points.

The seven use-after-freing errors could cause non-deterministic bugs if another thread re-allocated the freed memory. The most common case was five cut-and-paste uses of the code:

```
/* drivers/isdn/pcbit:pcbit_init_dev */
kfree(dev);
iounmap((unsigned char*)dev->sh_mem);
release_mem_region(dev->ph_mem, 4096);
```

Here, the memory pointed to by `dev` is freed and then immediately used in two subsequent function calls.

Additionally, the checker discovered two under-allocation errors. These were particularly dangerous, since they could cause memory corruption whenever a routine is used, rather than only failing under high load. One was caused by an apparent typo where the size of the memory needed for a structure of type `struct atm_mpoa_qos` (92 bytes) was computed using the size of a structure of type `struct atm_qos` (84 bytes):

```
/* net/atm/mpc.c:169:atm_mpoa_add_qos */
struct atm_mpoa_qos *entry;
...
entry = kmalloc(sizeof(struct atm_qos),
    GFP_KERNEL);
```

The other error reversed `kmalloc`'s size and interrupt level arguments, specifying that 7 (the value of `GFP_KERNEL`) bytes of storage to be allocated instead of 16. Currently, both errors are harmless, since the kernel uses a power-of-two memory allocator with a minimum allocation unit of 32 bytes. However, they are latent time bombs if a more space efficient allocator is ever used.

```
sm null_checker {
    decl { scalar } sz;          // match any scalar
    decl { const int } retv;     // match const ints
    decl { any_ptr } v1;        // match any ptr
    // 'state' specifies 'v' will have a state
    state decl { any_ptr } v;

    // Associate allocated memory with unknown
    // state until compared to null.
    start, v.all:
        // set v's state on true path to "null",
        // on false path to "not_null"
        { ((v = (any)malloc(sz)) == 0) }
        ==> true=v.null, false=v.not_null
        // vice versa
        | { ((v = (any)malloc(sz)) != 0) }
        ==> true=v.not_null, false=v.null
        // unknown state until observed.
        | { v = (any)malloc(sz) } ==> v.unknown;

    // Allow comparisons on variables in
    // states "unknown", "null", and "not_null."
    v.unknown, v.null, v.not_null:
        { (v == 0) } ==>
            true = v.null, false = v.not_null
        | { (v != 0) } ==>
            true = v.not_null, false = v.null;

    // Catch error path leaks by warning when
    // a non-null, non-freed variable gets to a
    // return of a negative integer.
    v.unknown, v.not_null: { return retv; } ==>
        { if(mgk_int_cst(retv) < 0)
            err("Error path leak!"); };

    // No dereferences of null or unknown ptrs.
    v.null, v.unknown: { *(any *)v } ==>
        { err("Using ptr illegally!"); };

    // Allow free of all non-freed variables.
    v.unknown, v.null, v.not_null:
        { free(v); } ==> v.freed;

    // Check for double free and use after free.
    v.freed:
        { free(v) } ==> { err("Dup free!"); }
        | { v } ==> { err("Use-after-free!"); };

    // Overwriting v's value kills its state
    v.all: { v = v1 } ==> v.ok;
}
```

Figure 4: *Metal* extension that checks that allocated memory is (1) checked before use, (2) not used after a free, (3) not double freed, and (4) always freed on error paths (those returning a negative integer).

```

/* from drivers/char/tea6300.c */
static int tea6300_attach(...) {
    ...
    client = kmalloc(sizeof *client, GFP_KERNEL);
    if (!client)
        return -ENOMEM;
    ...
    tea = kmalloc (sizeof *tea, GFP_KERNEL);
    if (!tea)
        return -ENOMEM;
    ...
    MOD_INC_USE_COUNT;
    ...
}

```

Figure 5: Code with two errors: (1) not freeing memory (`client`) on an error path and (2) (discussed in Section 6) calling `MOD_INC_USE_COUNT` after potentially blocking memory allocation calls.

While these checks focus on raw byte memory management, the general extension template can be retrofitted to check similar rules for other, higher-level objects. A modified version of this extension found 15 probable errors in Linux “IRQ” allocation code where allocations were not checked for errors, and IRQ’s were not deallocated on error paths.

6 Enforcing Rules Globally

The extensions described thus far have been implemented as local analyses. However, many systems rules are context dependent and apply globally across functions in a given call chain. This section presents two extensions that use *xg++*’s global analysis framework to check the following Linux rules:

1. Kernel code cannot call blocking functions with interrupts disabled or while holding a spin lock. Violating this rule can lead to deadlock [28].
2. A dynamically loaded kernel module cannot call blocking functions until the module’s reference count has been properly set. Violating this rule leads to a race condition where the module could be unloaded while still in use [26].

We first describe a global analysis pass that computes a transitive closure of all potentially blocking routines. Then, we discuss how the two extensions use this result.

6.1 Computing blocking routines

We build a list of possibly blocking functions in two passes. The first, local pass, is a *metal* extension that

Check	Local	Global	False Pos
Interrupts	18	42	4
Spin Lock	21	42	4
Module	22	~ 53	~ 2
Total	61	~ 137	~ 10

Table 3: Results for checking if kernel routines block (1) with interrupts disabled (“Interrupts”), (2) while holding a spin lock (“Spin Lock”), or (3) in a way that causes a module race (“Module”). We divide errors into whether they needed local or global analysis. Local errors were due to direct calls to blocking functions; global errors reached a blocking routine via a multi-level call chain. The global analysis results for Module are marked as approximate since they have not been manually confirmed.

traverses over every kernel routine, marking it if it calls functions known to potentially block. In Linux, blocking functions are primarily (1) kernel memory allocators called without the `GFP_ATOMIC` flag (which specifies not to sleep when the request cannot be fulfilled) or (2) routines to move data to or from user space (these block on a page fault). After processing each routine, the extension calls *xg++* support routines to emit the routine’s flow graph to a file. The flow graph contains (1) the routine’s annotation (if any) and (2) all procedures the routine calls. After the entire kernel has been processed, each input source file will have a corresponding emitted flow graph file. The second, global pass, uses *xg++* routines to link together all these files into a global call graph for the entire kernel. The global pass then uses *xg++* routines to perform a depth first traversal over this call graph calculating which routines have any path to a potentially blocking function. The output of this pass is a text file containing the names of all functions that could ever call a blocking function. Running the global analysis on the Linux kernel gives roughly 3000 functions that could potentially sleep.

6.2 Checking for blocking deadlock

Linux, like many OSes, uses a combination of interrupt disabling and spin locks for mutual exclusion. Interrupt disabling imposes an implicit rule: a thread running with interrupts disabled cannot block, since if it was the last runnable thread, the system will deadlock. Similarly, because of the implementation of Linux kernel thread scheduling, threads holding spin locks cannot block. Doing so causes deadlock when a sleeping thread holds a spin lock that a thread on

the same CPU is trying to acquire.

Our *metal* extension checks both rules by assuming each routine starts in a “clean” state with interrupts enabled and no locks held. As it traverses each code path, if it hits a statement that disables interrupts, it goes to a disabled state; an enable interrupt call returns it to the original state. Similarly, if it hits a function that acquires a spin lock, it traverses to a locked state; an unlock call returns it to the clean state. While in either of these states (or their composition), the extension examines all function calls and reports an error if the call is to a function in the list of potentially blocking routines.

Despite the simplicity of these rules, real code violates it in numerous places. The extension found 123 errors in Linux. Of those errors, 79 could lead to deadlock. The remaining 44 were calls to `kmalloc` with interrupts disabled. Possibly motivated by the frequency of this error, the `kmalloc` code checks if it is called with interrupts disabled, and, if so, it prints a warning and re-enables interrupts. In situations where interrupt disabling was used for synchronization, this leads to race conditions. The following code snippet is representative of a typical error (the mistake has been annotated in the source but not fixed):

```
/* drivers/sound/midibuf.c */
save_flags(flags);
cli();
...
while (c < count)
{
    ...
    for (i = 0; i < n; i++)
        /* BROKE BROKE-CANT DO THIS WITH CLI!! */
        copy_from_user((char *)&tmp_data,
                        &(buf)[c], 1);
        QUEUE_BYTE(midi_out_buf[dev], tmp_data);
        c++;
}
restore_flags(flags);
```

The call to `copy_from_user` can implicitly sleep, but is called after interrupts have been disabled with the call to `cli`.

The local errors seem to be caused by driver implementors not having a clear picture of either (1) the rules they have to follow and (2) that user data movement routines can block. The global errors seem to be caused by the fact that it is often hard to tell if a function can potentially block without tediously tracing through several function calls in different files, or without a considerable amount of *a priori* Linux kernel knowledge.

The checker produced eight false positives. Six were because the global calculation of blocking functions does not check if a called function would re-enable interrupts before calling a blocking function.

Two others were caused by name conflicts where a file defined and called a function with the same name as a blocking function.

The approach of this section also applies to other operating systems. Another implementor used our system to write an extension for the OpenBSD system that checked if interrupt handling code called a blocking operation. He found one bug where an interrupt handler could call a page allocation routine that in turn called a blocking memory allocator [29].

6.3 Checking module reference counts

Linux allows kernel subsystems to be dynamically loaded and unloaded. Modules have a reference count tracking the number of kernel subsystems using them. Modules increment this count during loading (using `MOD_INC_USE_COUNT`) and decrement it during unloading (using `MOD_DEC_USE_COUNT`). The kernel can unload modules with a zero reference count at any time. A module must protect against being unloaded while sleeping by incrementing its reference count before calling a blocking function. Similarly, during unloading, it cannot block after decrementing its count. Finally, if the module aborts installation after incrementing its reference count, it must decrement the count to restore it to its original value.

Our extension checks for load race conditions by tracking if a potentially blocking function has been called and flagging subsequent `MOD_INC`s. Conversely, it checks for unload race conditions by tracking if a `MOD_DEC` has been performed and flagging subsequent calls to potentially blocking functions. It finds dangling references by emitting an error when a `MOD_INC` has not been reversed along a path that returns a negative integer (which idiomatically signals an error). As Table 3 shows, a local version of the extension that did not use the global list of blocking functions found 22 rule violations, whereas the global version found 53 cases (we have not yet confirmed the global errors).

7 Linux Mutual Exclusion

The complexity of dealing with concurrency leads most of the Linux kernel and its device drivers to follow a localized strategy where critical sections begin and end within the same function body. Despite this stylized use, the size of the code and implementors’ imperfect understanding leads to errors. We wrote an extended version of the interrupt checker described in Section 3 to check that each kernel function conforms to the following conditions:

Condition	Applied	Bug	False Pos
Holding lock	~ 5400	29	113 (90)
Double lock	-	1	3
Double unlock	-	1	20 (18)
Intr disabled	~ 5800	44 (43)	63 (54)
Bottom half	~ 180	4	12
Bogus flags	~ 3200	4	49 (24)
Total	-	83 (82)	260 (201)

Table 4: Results of running the Linux synchronization primitives checker on kernel version 2.3.99. The **Applied** column is an estimate of the number of times the check was applied. We skipped twelve warnings that were difficult to classify. The parenthesized numbers show the changes when the two files with the most false positives are ignored.

1. All locks acquired within the function body are released before exiting.
2. No execution paths attempt to lock or unlock the same lock twice.
3. Upon exiting, interrupts are either enabled or restored to their initial state.
4. The “bottom halves” of interrupt handlers are not disabled upon exiting.
5. Interrupt flags are saved before they are restored.

Table 4 shows the results of running the extension on Linux. The “Applied” column is an estimate of the number of times each check was applied. Two device drivers account for a large number of false positives because they use macros that consult runtime state before locking or unlocking. The parenthesized numbers show the changes in the false positive results (over 20%) when these two files are ignored.

The most common bugs are either holding a lock or leaving interrupts disabled on function exit. These bugs often occur when detecting an error condition after which the function returns immediately. For example, the checker found this bug in a device driver for PCMCIA card services

```
/* drivers/pcmcia/cs.c:
    pcmcia_deregister_client */
spin_lock_irqsave(&s->lock, flags);
client = &s->clients;
while ((*client) && ((*client) != handle))
    client = &(*client)->next;
if (*client == NULL)
    /* forgot about &s->lock, flags! */
    return CS_BAD_HANDLE;
```

The checks for Linux locking conventions have resulted in seven kernel patches, including a fix for the error shown above. All seven patches fix cases where a lock is mistakenly held when exiting a function, and six of the seven are in device drivers (the last patch was to an implementation of ipv4 network filters). We have not been able to confirm many of the other potential bugs with kernel or device driver developers, though several strong OS implementors have examined them and consider them to be at least suspicious. Most of the potential bugs are in device drivers and networking code – this is not surprising since much of this code is written by developers throughout the world with varying degrees of familiarity with the Linux kernel.

The false positives mostly come from three sources. Code that intentionally violates the convention for the sake of efficiency or modularity accounts for 90 false positives. For example, sometimes a family of related device drivers will define an interface that breaks the conventions. Another large source of false positives (48) is caused by the fact that our checker only performs local analysis. Some drivers implement their own locking functions using the basic primitives provided by the system. The checker will warn when these functions exit holding a lock or with interrupts disabled, which is exactly what they are supposed to do. Global analysis could eliminate many of these false positives. Finally, the fact that our system does not prune simple, impossible paths accounts for 35 false positives. A typical example of this is when kernel code conditionally acquires a lock, performs an action, and then releases the lock based on the same condition. There are only two possible paths through this code, not the four that our system thinks exist.

The remaining 21 false positives could be eliminated by extending the checker’s notion of locking functions and changing our system to prune the false branch of loop conditionals of the form “for(;;).”

8 Optimizing FLASH

In addition to checking, MC can be used for optimization. Below, we describe three extensions written to find system-level optimization opportunities in the FLASH machine’s cache coherence code [20]. This code must be fast because it implements functionality (cache coherence) that is usually placed in hardware. Eliminating even a single instruction is considered beneficial. Several of the protocols examined here have been aggressively tuned for years due to their use in numerous performance papers as evidence for the effectiveness of software-controlled

Optimization	Number	False Pos	LOC
Buffer Free	11	9	30
Message Length	40	0	32
XOR Opcode	hundreds	~10	400(*)

Table 5: MC-based FLASH optimizer results. **Number** counts how many optimization opportunities were found. The XOR checker is written in an old version of the system — a version written in *metal* would be several factors smaller.

cache coherence. Despite this effort, MC optimizers found hundreds of optimization opportunities, mostly due to the difficulty in manually performing equivalent searches across FLASH’s deeply nested paths.

Buffer-free optimization. Each time a FLASH node receives a message, it invokes a customized message protocol handler that determines how to satisfy the request and update the protocol state. Handlers use the incoming message buffer to send outgoing data messages, and must free it before exiting. Handlers can send data messages, which need a buffer, and control messages, which do not. Many handlers send more than one message when responding to a request. To minimize the chance of losing a buffer, implementors are typically conservative and defer buffer freeing until the last handler send, irrespective of whether the last send(s) was a control message and therefore did not need a buffer. Unfortunately, while this strategy simplifies handler code, it increases buffer contention under high load.

Our extension indicates when buffer frees can occur earlier in the code. It traces all sends on each path through the function, and by looking at send arguments, detects if the send (1) needs a buffer and (2) frees its buffer. It gives a suggestion for any path that has an active buffer that ends with a “suffix” of control sends. The extension is 56 lines long, and found 11 instances in a large FLASH protocol, “*dyn_ptr*,” where the buffer could be safely freed earlier. Each of these optimizations could be implemented by changing only two lines of code. The extension also produced nine false positives. Most of these were cases where the execution path was too complex to optimize without major code restructuring.

Redundant length assignments. Our second, lower-level optimization extension detects redundant assignments to a message buffer’s length field. For speed, when sending multiple messages, implementors set a buffer’s message length early in a handler and then try to reuse this setting across multiple messages. Long path lengths make it easy to miss redundant assignments. Our checker detects redundancies

by recording the last assignment on every path and warning if there are two assignments of the same constant. It discovered 40 redundant assignments in the FLASH protocol code.

Efficient opcode setting. Message headers must specify the message’s opcode (type). Opcode assignment costs two instructions. However, if the handler knows what opcode is currently in a header, it can change the opcode in one instruction by xoring the message header with the xor of the new and current opcode. Our extension detects such cases by computing when a message header, with known opcode, is assigned a new opcode. Both the old and new opcodes must be the same on all incoming paths. The extension determines the initial header value by looking in an automatically-built list of all opcodes a handler might receive. If there is only one possible opcode value, the extension records it and starts in a “known” state. Otherwise, the checker starts in an “unknown” state. It transitions from this state to the “known” state after the first opcode assignment. Each assignment encountered in the known state is annotated with the current opcode value. A second pass then checks every assignment and, if all paths reached it in the known state with the same opcode, emits a warning to the user that xor could be used to save an instruction. This checker found hundreds of such cases.

9 Conclusion

Systems are pervaded with restrictions of what actions programmers must always or never perform, how they must order events, and which actions are legal in a given context. In many cases, these restrictions link together the entire system, creating a fragile, intricate mess. Currently, systems builders obey these restrictions as well as they can. Unfortunately, system complexity makes such obedience difficult to sustain. Programmers make mistakes, and often they have only an approximate understanding of important system restrictions. Such mistakes can easily evade testing, which rarely exercises all cases.

We have shown that many system restrictions can be automatically checked and exploited using meta-level compilation (MC). MC makes it easy for implementors to extend compilers with lightweight system-specific checkers and optimizers. Currently, a system rule must be understood by all implementors. MC allows one implementor, who understands this rule, to write a check that is enforced on everyone’s code. This leverage exerts tremendous practical force on the development of complex systems.

Check	Errors	False Positives	Uses	LOC
Side-effects (§ 4.1)	14	2	199	25
Static assert (§ 4.2)	5	0	1759	100
Stack check (§ 4.3)	10+	0	332K	53
User-ptr (§ 5.1)	18	15	187	68
Allocation (§ 5.2)	184	64	4732	60
Block (§ 6.2)	123	8	-	131
Module (§ 6.3)	~75	2	-	133
Mutex (§ 7)	82	201	14K	64
Total	~511	~292	-	669

Table 6: The results of MC-based checkers summarized over all checks. **Error** is the number of errors found, **False Positives** is the number of false positives, **Uses** is the number of times the check was applied, and **LOC** is the number of lines of *metal* code for the extension (including comments and whitespace).

MC is a general approach, scaling from simple cases such as checking assertions up to global strategies for mutual exclusion and deadlock avoidance. We have demonstrated MC's power by using it to check four real, heavily-used, and tested systems. It found bugs in all of them — roughly 500 in all — many of which would be difficult to find with testing or manual inspection. Further, these extensions typically required less than a day and a hundred lines of code to implement. Curiously, writing code to check restrictions is significantly easier than writing code that obeys them. With few exceptions, our extensions were written by programmers who, at best, only had a passing familiarity with the systems to which they were applied. We believe that these results show that the use of meta-level compilation can significantly aid system construction.

10 Acknowledgements

We thank David Dill for many discussions on software checking, Wilson Hsieh for discussions about the initial *metal* language, Mark Heinrich for his willingness to validate our FLASH results. Wallace Huang and Yu Ping Hu verified many of the error messages from the null checker in Section 5. Rusty Russell, Tim Waugh, Alan Cox, and David Miller answered numerous Linux questions and gave feedback on error reports. Additionally, we thank the other generous readers of *linux-kernel* for their feedback and support. We thank Mark Horowitz for his support, and the initial suggestion of looking at FLASH, which led to numerous other interesting directions. Finally, we thank David Dill, Wilson Hsieh, Mark Horowitz, the anonymous reviewers and especially Andrew Myers for valuable feedback on the paper.

References

- [1] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing systems*, pages 131–152, Spring 1996.
- [2] R. S. Boyer and Y. Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 1(43):166–192, January 1996.
- [3] S. Chiba. A metaobject protocol for C++. In *OOP-SLA 1995 Conference Proceedings Object-oriented programming systems, languages, and applications*, pages 285–299, October 1995.
- [4] A. Chou, B. Chelf, D.R. Engler, and M. Heinrich. Using meta-level compilation to check FLASH protocol code. To appear in ASPLOS 2000, November 2000.
- [5] A. Chou and D.R. Engler. Metal: A language and system for building lightweight, system-specific software checkers, analyzers and optimizers. Available upon request: acc@cs.stanford.edu, 2000.
- [6] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *ICSE 2000*, 2000.
- [7] R. F. Crew. ASTLOG: A language for examining abstract syntax trees. In *Proceedings of the First Conference on Domain Specific Languages*, pages 229–242, October 1997.
- [8] D.L. Detlefs, R.M. Leino, G. Nelson, and J.B. Saxe. Extended static checking. TR SRC-159, COMPAQ SRC, December 1998.
- [9] D.R. Engler. Incorporating application semantics and control into compilation. In *Proceedings of the First Conference on Domain Specific Languages*, October 1997. An extended version “Interface Compilation: Steps toward Compiling Program Interfaces as Languages” was selected to appear in *IEEE Transactions on Software Engineering*, May/June, 1999, Volume 25, Number 3, p 387–400.

- [10] D. Evans, J. Guttag, J. Horning, and Y.M. Tan. Lclint: A tool for using specifications to check code. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, December 1994.
- [11] R. W. Floyd. *Assigning meanings to programs*, pages 19–32. J.T. Schwartz, Ed. American Mathematical Society, 1967.
- [12] C. W. Fraser and D. R. Hanson. *A retargetable C compiler: design and implementation*. Benjamin/Cummings Publishing Co., Redwood City, CA, 1995.
- [13] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, December 1992.
- [14] G. Holzmann and M. Smith. Software model checking: Extracting verification models from source code. In *Invited Paper. Proc. PSTV/FORTE99 Publ. Kluwer*, 1999.
- [15] Intrinsa. A technical introduction to PREFIX/Enterprise. Technical report, Intrinsa Corporation, 1998.
- [16] M.F. Kaashoek, D.R. Engler, G.R. Ganger, H.M. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, October 1997.
- [17] G. Kiczales, J. des Rivieres, and D.G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
- [18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, June 1997.
- [19] A. Kolawa and A. Hicken. Insure++: A tool to support total quality software. www.parasoft.com/insure/papers/tech.htm.
- [20] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994.
- [21] J. Lamping, G. Kiczales, L.H. Rodriguez Jr., and E. Ruf. An architecture for an open compiler. In *Proceedings of the IMSA'92 workshop on reflection and meta-level architectures*, 1992.
- [22] T. Lord. Application specific static code checking for C programs: Ctool. In *twaddle: A Digital Zine (version 1.0)*, 1997.
- [23] K.L. McMillan and J. Schwalbe. Formal verification of the gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 242–51. Tokyo, Japan Inf. Process. Soc., 1991.
- [24] T.C. Mowry, A.K. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, 1996.
- [25] G. Nelson. *Techniques for program verification*. Available as Xerox PARC Research Report CSL-81-10, June, 1981, Stanford University, 1981.
- [26] K. Owens. "Please review all modules for unload races". Sent to linux-kernel@vger.rutgers.edu. Gives protocol to follow to prevent module unload races., 2000.
- [27] R. Rashid. Personal communication. Microsoft's internal tool used to check violations in Windows device drivers., July 2000.
- [28] P. Russell (rusty@linuxcare.com). Unreliable guide to hacking the Linux kernel. Distributed with the 2.3.99 Linux RedHat Kernel, 2000.
- [29] C.P. Sapuntzakis. Personal communication. Bug in OpenBSD where an interrupt context could call blocking memory allocator, April 2000.
- [30] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T.E. Anderson. Eraser: A dynamic data race detector for multithreaded programming. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.
- [31] A. Srivastava and A. Eustace. ATOM — a system for building customized program analysis tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, 1994.
- [32] U. Stern and D.L. Dill. Automatic verification of the SCI cache coherence protocol. In *Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings*, 1995.

Devil: An IDL for Hardware Programming

Fabrice Méryllon Laurent Réveillère*
Charles Consel* Renaud Marlet† Gilles Muller

Compose Group, IRISA / INRIA, University of Rennes I
Campus Universitaire de Beaulieu, F-35042 Rennes Cedex, France

E-mail: {merillon,lreveill,consel,marlet,muller}@irisa.fr

Abstract

To keep up with the frantic pace at which devices come out, drivers need to be quickly developed, debugged and tested. Although a driver is a critical system component, the driver development process has made little (if any) progress. The situation is particularly disastrous when considering the hardware operating code (i.e., the layer interacting with the device). Writing this code often relies on inaccurate or incomplete device documentation and involves assembly-level operations. As a result, hardware operating code is tedious to write, prone to errors, and hard to debug and maintain.

This paper presents a new approach to developing hardware operating code based on an Interface Definition Language (IDL) for hardware functionalities, named Devil. This IDL allows a high-level definition of the communication with a device. A compiler automatically checks the consistency of a Devil definition and generates efficient low-level code.

Because the Devil compiler checks safety critical properties, the long-awaited notion of robustness for hardware operating code is made possible. Finally, the wide variety of devices that we have already specified (mouse, sound, DMA, interrupt, Ethernet, video, and IDE disk controllers) demonstrates the expressiveness of the Devil language.

*Author's current address: LaBRI / ENSERB, 351 cours de la Libération, F-33405 Talence Cedex, France.

†Author's current address: Trusted Logic, 5 rue du Bailliage, F-78000 Versailles, France. E-mail: Renaud.Marlet@trusted-logic.fr.

1 Introduction

A device driver is a key system component that makes hardware innovation available to end users. Device drivers are critical both in general-purpose computers and in the fast-evolving domain of appliances. If driver development falls behind, product competitiveness can be compromised. If a device driver is faulty, a hardware innovation may turn into a disaster instead of improving competitiveness.

Still, ever since the first device drivers have been written, their development process has made little (if any) progress. This situation has particularly disastrous effects when considering hardware operating code (i.e., code communicating with the hardware). This layer of code is well-known to be *low level* and *error prone*.

Hardware operating code is low level because it consists of many bit operations. Indeed, we have found that bit operations can represent up to 30% of driver code¹. Such low-level programming is obviously prone to errors and requires tedious debugging. In fact, advances in programming languages have had no impact on the development of hardware operating code: there is no syntactic support for low-level operations, there is no verification support to identify incorrect usage of these operations, and there is no tool support to facilitate debugging.

Additionally, hardware documentation typically contains imprecise or inaccurate information. Therefore, writing hardware operating

¹This measurement was performed on various Linux 2.2-12 drivers.

code typically involves laboriously searching for obscure incantations aimed at performing specific operations on the device. Not only can this sometime cause unexpected behavior, but it also makes re-use of hardware operating code difficult.

Finally, there are no recognized methodologies for structuring device drivers. Even worse, a driver is often written by modifying an existing one. As a result, the code quickly becomes tangled, which makes debugging and maintenance complex.

Our proposal

This paper describes a new approach to developing the hardware operating layer of a driver. Our approach allows drivers to be written in a high-level language, allows important safety properties to be checked, and allows low-level code to be automatically generated.

We introduce an Interface Definition Language (IDL) to describe hardware functionalities, named Devil. IDLs are extensively used in modern OSes, either to hide heterogeneity and intricacies of message construction in distributed systems [3, 13], or to glue together components in modular operating systems [2, 9, 10]. Just as RPC IDLs conventionally define operations and their input/output types, Devil specifies the functional interface of the device. To do so, it provides the programmer with abstractions and syntactic constructs that are specific to describing devices. From a Devil specification, a compiler automatically generates stubs containing low-level code to operate the device. Furthermore, verification tools enable critical safety properties to be checked at compile time, and at run time if necessary.

Just as an IDL typically allows code to be re-used, a Devil specification can be re-used in different contexts (*e.g.*, various operating systems). More generally, our vision is that Devil specifications either should be written by device vendors or should be widely available as public domain libraries in order to ease driver development.

Our contributions are as follows.

- We have designed and implemented an IDL for devices. This language is an alternative to assembly-language-like programming of devices.
- We propose tools to verify critical safety properties of hardware operating code. These tools enable us to provide the long-awaited notion of *robustness* for device drivers.
- We present a comparison between Devil specifications and existing driver code. This comparison is based on experimental data which demonstrate that a Devil specification is up to 5.9 times less prone to errors than C code, with almost no loss in performance.

The rest of this paper is organized as follows. Section 2 presents the Devil language. Section 3 describes the safety properties that can be verified both statically on Devil specifications and dynamically by the generated interface. Section 4 assesses the benefits of our approach by comparing hand-crafted drivers with equivalent ones written using Devil. Section 5 describes related work. Section 6 concludes and suggests future work.

2 Devil

Devil is an IDL for specifying the functional interface of a device. To design Devil, we have studied a wide spectrum of devices and their corresponding drivers, mainly from Linux sources: Ethernet, video, sound, disk, interrupt, DMA and mouse controllers. This study was supported by literature about driver development [7, 16], device documentation available on the web, and discussions with device driver experts for Windows, Linux and embedded operating systems. Devil has proved expressive enough to describe even devices having a convoluted interface such as the Crystal CS4236B sound controller.

Concretely, a device can be described by three layers of abstraction: *ports*, *registers*, and *device variables*. The entry point of a Devil specification is the declaration of a device, parameterized by *ports* or ranges of ports, which

```

device logitech_busmouse (base : bit[8] port @ {0..3})      1
{
    // Signature register (SR)
    register sig_reg = base @ 1 : bit[8];                    4
    variable signature = sig_reg, volatile, write trigger : int(8); 5

    // Configuration register (CR)
    register cr = write base @ 3, mask '1001000.' : bit[8];    8
    variable config = cr[0] : { CONFIGURATION => '1', DEFAULT_MODE => '0' }; 9

    // Interrupt register
    register interrupt_reg = write base @ 2, mask '000.0000' : bit[8]; 12
    variable interrupt = interrupt_reg[4] : { ENABLE => '0', DISABLE => '1' }; 13

    // Index register
    register index_reg = write base @ 2, mask '1..00000' : bit[8]; 16
    private variable index = index_reg[6..5] : int(2); 17

    register x_low = read base @ 0, pre {index = 0}, mask '****...' : bit[8]; 19
    register x_high = read base @ 0, pre {index = 1}, mask '****...' : bit[8]; 20
    register y_low = read base @ 0, pre {index = 2}, mask '****...' : bit[8]; 21
    register y_high = read base @ 0, pre {index = 3}, mask '...*...' : bit[8]; 22

    structure mouse_state = {
        variable dx = x_high[3..0] # x_low[3..0], volatile : signed int(8); 24
        variable dy = y_high[3..0] # y_low[3..0], volatile : signed int(8); 25
        variable buttons = y_high[7..5], volatile : int(3); 26
    }; 27
}

```

Figure 1: Logitech Busmouse Specification

abstract physical addresses. Ports then allow device *registers* to be declared; these define the granularity of interactions with the device. Finally, *device variables* are defined from registers, forming the functional interface to the device.

These three layers of abstraction are illustrated by the following fragment of the Devil description of the Logitech Busmouse controller (see Figure 1 for a complete description).

```

device logitech_busmouse(base : bit[8] port@{0..3})
{
    register sig_reg = base @ 1 : bit[8];
    variable signature = sig_reg, ... : int(8);
    ...
}

```

The `logitech_busmouse` declaration is parameterized by a range of ports specified as the main address `base` and a range of offsets (from 0 to 3). An eight-bit register `sig_reg` is declared at port `base`, offset by 1. Finally, the device variable `signature` is the interpretation of this register as an eight-bit unsigned integer. This fragment declares a device whose functional interface consists of a device variable (`signature`). Only device variables are visible from outside a Devil description ports and registers are hidden. In fact, for each variable the Devil compiler generates two C stubs that per-

mit to write or read the variable by emitting the proper I/O operations.

In the rest of this section, we first describe the basic Devil constructs, and then present advanced Devil features that allow the description of devices with contorted addressing modes.

2.1 Basic Devil

Ports, registers, and device variables are the basic layers of abstraction that describe the interface of a device. We now present their usage by describing in detail the Devil specification of the Logitech Busmouse (see Figure 1), and a fragment of the NE2000 Ethernet controller.

Ports. The port abstraction is at the basis of the communication with the device. A port hides the fact that, depending on how the device is mapped, it can be operated via either I/O or memory operations. A device often has several communication points whose addresses are derived from one or more base addresses. Therefore, the port constructor, denoted by `@`, takes as arguments a ranged port and a constant offset (*e.g.*, `base@1` as illustrated by line 4 of the Busmouse specification). To enable veri-

fication, the range of valid offsets must be specified within the entry point declaration (e.g., `port@{0..3}` as illustrated by line 1 of the Busmouse specification).

Registers. Registers define the granularity of interaction with a device; as such register size (in number of bits) must be explicitly specified. Registers are typically defined using two ports: one for reading and one for writing. Only one port needs to be provided when reading and writing share the same port, or when the register is read-only or write-only.

A register declaration may be associated with a mask to specify bit constraints. An element of this mask can either be '*' to denote a relevant bit, '0' or '1' to denote a bit that is irrelevant when read but has a fixed value (0 or 1) when written, or '-' to denote a bit that is irrelevant whether read or written. As an example, consider the declaration of the write-only register `index_reg` in line 16 of the Busmouse specification.

```
register index_reg =
  write base@2, mask '1..00000' : bit[8];
```

This mask indicates that only bits 6 and 5 are relevant. Also, bit 7 is forced to 1 when written while bits 4 through 0 are forced to 0. Proper register masking is performed as part of the stubs generated by the Devil compiler.

Device variables. In order to minimize the number of I/O operations required for communicating with a device, hardware designers often group several independent values into a single register. Accessing these values requires bit mask and shift operations which are error-prone in a general programming language such as C. Devil abstracts values as device variables, which are defined as a sequence of bit registers. Device variables are strongly typed in order to detect potential misuses of the device. Possible types are booleans, enumerated types, signed or unsigned integers of various sizes, and ranges or sets of integers. In line 17 of the Busmouse specification, the 5th and 6th bit of the `index_reg` register make up a two-bit unsigned integer variable (i.e., a variable that can take a value from 0 to 3). The private attribute

means that the `index` variable is not defined in the functional interface of the Busmouse controller and can not be directly accessed by the driver programmer.

```
private variable index = index_reg[6..5] : int(2);
```

Access pre-actions. Device functionalities are often extended by mapping multiple registers to a single physical address. Examples are index-based addressing mode and banks of registers. As a result, accessing such registers requires the setting of a specific context which may involve several I/O operations. To capture this situation, Devil allows pre-actions to be attached to a register. Lines 19 and 20 of the Busmouse specification declare two read-only registers on the same port `base@0`, provided that the variable `index` is set either to 0 or 1 prior to the port access.

```
register x_low = read base@0, mask '****...',
  pre {index = 0} : bit[8];
register x_high = read base@0, mask '****...',
  pre {index = 1} : bit[8];
```

Register concatenation. Device variables can be spread over several registers. As illustrated by line 25 of the Busmouse specification, constructing the `dx` variable requires concatenation of the two registers `x_high` and `x_low`. The 8-bit variable `dx` is obtained by concatenating the four lower bits of register `x_high` with the four lower bits of register `x_low`.

```
variable dx = x_high[3..0] # x_low[3..0], ...
```

Enumerated types. Devil allows defining an enumerated type to abstract the concrete representation of bit values. The symbols `<=`, `=>` and `<=>` define read, write and read-write constraints, respectively. Enumerated types are used to specify the valid values of a device variable. As an example, the `config` variable declaration shown in line 9 of the Busmouse specification declares the two modes (`CONFIGURATION` and `DEFAULT_MODE`) that can be written to the `config` variable.

```
variable config = cr[0] : {
  CONFIGURATION => '1', DEFAULT_MODE => '0' };
```

Caching and synchronization. Sharing one or more registers between variables induces cache and synchronization problems. When one variable needs to be written independently

from the others, the Devil compiler has to determine a value to assign to the other variables. The choice of value depends on whether the access to that variable is idempotent. A Devil variable can be associated with a *behavior* qualifier that specifies the access semantics. No qualifier (the default case) means that the access is idempotent and thus can be redone without side effect; consequently, the variable value can be cached. Such a behavior is often associated with variables that serve as parameters.

A **trigger** behavior means that a write (or read) access to the variable induces a side effect on the controller. Since the side effect cannot be re-done, multiple trigger variables cannot be defined on a register unless a neutral value is provided. Command variables usually have a trigger behavior. The following fragment from an NE2000 Ethernet controller presents examples of the trigger behavior.

```
register cmd = base@0 : bit[8];
variable st = cmd[1..0],
    write trigger except NEUTRAL;
variable txp = cmd[2],
    write trigger except NOP;
variable rd = cmd[5..3],
    write trigger except NODMA;
private variable page = cmd[7..6] : int(2);
```

In this example, the register `cmd` is split into four variables. While the `page` variable has an idempotent behavior, the variables `st`, `txp` and `rd` trigger an action when written, except for specific values (`NEUTRAL`, `NOP` and `NODMA`).²

Finally, a **volatile** behavior specifies that a read operation is not idempotent; two successive reads may deliver different values. When one needs to get a consistent value of several volatile variables, it is necessary to read them together in one or multiple read operations and cache the result for later use. To do so, Devil allows several variables to be grouped using a **structure**. The use of a structure is demonstrated by the `dx`, `dy` and `buttons` variables of the Busmouse specification (lines 19 to 22).

```
structure mouse_state = {
    variable dx =
        x_high[3..0] # x_low[3..0], volatile : ...
    variable dy =
        y_high[3..0] # y_low[3..0], volatile : ...
    variable buttons = y_high[7..5], volatile : ...
};
```

²These values are defined using an enumerated type, not shown here.

To access field variables `dy` and `buttons`, the programmer first has to read the `mouse_state` structure. Stubs generated for the structure perform the effective I/O operations, while stubs for the field variables access only the cache. It should be noted that since `dy` and `buttons` share the `y_high` register, `y_high` is read only once. Use of the stubs by the driver programmer is detailed in section 4.1.

Cache and synchronization issues are usually only informally documented by hardware vendors. When programming controllers in a general programming language, cache and synchronization issues are typically solved in an ad-hoc manner that limits code re-use and driver evolution. In fact, the lack of a rigorous description of variable behaviors often leads to laborious testing until the expected functionality is obtained. Also, without specific language support, no verification of the correct usage of variables is possible; this opens opportunities for undetected errors.

Assessment. By clearly defining the semantics of variable behavior, a Devil specification serves as knowledge repository for the correct use of a device. In fact, the driver programmer is guided by the interface generated from the Devil specification. This simplifies driver development and improves re-use. Furthermore, verification is possible at two design stages: (i) on the Devil specification itself so as to check consistency of declarations, (ii) on the correct usage of interface procedures generated by the Devil compiler. These advantages are even more crucial when the device interface is awkward and contorted. The next section presents advanced Devil constructions which permit to handle these situations.

2.2 Advanced Devil

To maximize performance, most modern devices offer a simple, flat interface to registers. However, devices are rarely built from scratch and many of them are evolutions or supersets of previous controllers. For example, today's PCs still rely on DMA, interrupt and graphics controllers that were designed more than twenty years ago.

Design constraints of older devices were guided not only by performance but also by technology and the size of the available I/O address space. Adding functionalities to a device while maintaining backward compatibility induces tricks for addressing additional registers. These issues result in contorted addressing modes, making the programming of such devices even more complex and error-prone. Devil has been specifically targeted towards supporting such devices. Let us now present some of the advanced Devil features using fragments from the Devil specifications of the 8237A DMA, the 8259A interrupt, the Crystal CS4236B, and the IDE controllers.

Register serialization. The 8237A DMA controller provides 16-bit counters through a single 8-bit port. As illustrated by the following example, constructing the counter `x` requires concatenation of the two registers `cnt_high` and `cnt_low`. Since these registers are accessed through the same port, a reading order has to be specified (`cnt_low` then `cnt_high`). Finally, a pre-action attached to `cnt_low` (write any value to the flip-flop variable) permits to reset an internal pointer to this register.

```
register cnt_low =
    data, pre {flip_flop = *} : bit[8];
register cnt_high = data : bit[8];
variable x = cnt_high # cnt_low : int(16)
    serialized as {cnt_low; cnt_high};
```

Control-flow based serialization. The 8259A interrupt controller possesses various execution modes that depend on the hardware configuration (processor type, cascaded/single controller) [12]. Initialization of the controller is performed by writing to configuration variables defined over four initialization registers. The initialization sequence varies with the actual values of configuration variables. Additionally, three of the configuration registers (e.g., `icw2`, `icw3`, `icw4`) are mapped to a single port and their addressing is implicitly done by previously written configuration values. The following example shows how such an addressing mode can be specified in Devil: configuration variables are grouped together within the `init` structure. Writing variables of this structure into registers is ordered using tests on variable values.

```
register icw1 =
    write base@0, mask '...1....' : bit[8];
register icw2 = write base@1 : bit[8];
register icw3 = write base@1 : bit[8];
register icw4 =
    write base@1, mask '000....' : bit[8];

structure init = {
    variable sngl = icw1[1] : {
        SINGLE => '1', CASCADED => '0' };
    variable ic4 = icw1[0] : bool;
    ...
    variable microprocessor = icw4[0] : {
        X8086 => '1', MCS80_85 => '0' };
} serialized as {
    icw1;
    icw2;
    if (sngl == SINGLE) icw3;
    if (ic4 == true) icw4;
};
```

Automata based addressing mode.

Among the chips we have studied, the Crystal CS4236B sound chip is one of the most complex. This chip is compatible with the Windows Sound System standard [5], but possesses 18 additional registers. These registers are doubly indexed through the I23 index. Writing a specific device variable converts I23 from an extended address register into an extended data register. To convert I23 back to an address register, the control register must be written. In order to specify this automata, Devil offers the notion of private variables that are not mapped to a specific register (`xm` in the following example). These variables can be used as memory cells and can be updated when writing a register or a device variable. The code below shows how the extended registers of the CS4236B can be specified using Devil.

```
private variable xm : bool;
register control =
    base@0, set {xm = false} : bit[8];
variable IA = control : int(0..31);

// Indexed Registers I0 - I31
register I(i : int(0..31)) =
    base@1, pre {IA = i} : bit[8];
register I23 = I(23), mask '.....0.';

variable ACF = I23[0] : bool;
structure XS = {
    variable XA = I23[2,7..4] : int(5);
    variable XRAE = I23[3], set {xm = XRAE},
        write trigger for true : bool;
};

// Extended Registers X0-X17,X25
register X(j : int(0..17,25)) = base@1,
    pre {XS = {XA=>j; XRAE=>true}} : bit[8];
```

Block transfer. On some processors, such as those of the Pentium family, replacing a

C loop over a variable read/write by a dedicated looping instruction (e.g., *rep* on the Pentium) is often more efficient. Variables with a block transfer usage have to be identified with a `block` keyword. For those variables, the Devil compiler generates two processor-specific block transfer stubs in addition to the single access stubs. The `Ide_data` variable declaration from the IDE specification shown below illustrates the use of the `block` attribute.

```
variable Ide_data =  
    ide_data, trigger, volatile, block : int(16);
```

Other features of Devil are not detailed here. These features include access post-actions, arrays, register constructors and conditional declarations depending on device modes. A complete description of Devil can be found in [17].

3 Property Verification

Devil has been designed to express domain-specific information about the functional interface of devices. Because this information is made explicit, Devil enables a variety of verifications that are beyond the scope of general programming languages. As a result, more errors can be caught earlier in the driver development process. In turn, debugging is easier and less time-consuming. Finally, the robustness of the driver is improved since the programmer has guarantees over the correctness of low-level interactions.

This section summarizes the properties that can be verified both when a Devil description is compiled and when the resulting interface implementation is used.

3.1 Verification of Devil specifications

Due to the declarative nature of the Devil language, it is possible to verify the following properties that ensure the consistency of a specification:

Strong typing. Devil abstractions (e.g., ports, registers, variables) are strongly typed: all uses of these abstractions can be matched

against their definition to check type correctness. Types describe usage constraints for registers and variables that are read or write only. Also, various size checks can be performed: the size of data accesses on ports, the size of registers, the size of variables derived from conversion functions, the size of bit masks, and the size of bit patterns that are associated a symbolic name in enumerated types, port ranges, and bit ranges for register fragments.

No omission. All declared entities in a Devil specification must be used at least once. This constraint concerns port arguments in a device declaration, values of ranged port offsets, registers, and register bits (although some bits can be declared irrelevant using bit masks). Read elements of a type mapping must be exhaustive. Also, a type for reading (as well as possibly writing) must be used with a readable variable. The same holds for writing.

No double definition. All entities in a Devil specification must be declared at most once. This constraint concerns port arguments in a device declaration, ports, registers, types, symbolic names and bit patterns in enumerated types and variables.

No overlapping definitions. Port and register descriptions must not overlap. More precisely, each port must appear only once in the register definitions, except when registers are defined using disjoint pre-actions or masks. However, the same port may be used for reading from one register and writing to another. No bit of a single register can be used in the definition of two different variables.

3.2 Verification of interface usage

Verification of the correct usage of the generated interface can be both static and dynamic. In the latter case, run-time checks are optionally included in the code for debugging purposes.

When writing to a variable, a check can be performed to verify that the written value falls

within the range specified by the variable type. If the value is constant, the check can generally be done at compile time. However, because the type system of C is not powerful enough to express all Devil types, not all such verifications can be implemented at compile time. In this situation, checks have to be implemented in debug mode using run-time checks. Finally, run-time checks can optionally be generated after variable reads. Such checks are useful for verifying that a device behaves accordingly to its Devil specification.

Our experience in re-engineering drivers showed that dynamic checks allow the early detection of usage errors, preventing them from becoming insidious bugs. This is particularly valuable for kernel-mode drivers, which are tricky to step through with a debugger. Moreover, since the checks are automatically and systematically inserted and removed by the compiler, their use is easy and safe.

4 Comparison with Hand-Crafted Drivers

To assess our approach, we now compare the use of Devil and C. First, we analyse issues related to code development. Then, we report on a study based on mutation analysis to evaluate the robustness of Devil and C implementations. Finally, we discuss the performance of drivers that use the C library automatically generated from a Devil specification.

4.1 Driver development

To illustrate the benefits of Devil in terms of separation of concerns and readability, we compare a fragment of the original C implementation of the Logitech Busmouse driver (see Figure 2) with the use of the interface (see Figure 3) generated from the equivalent Devil specification.

In a traditional C driver, the programmer writes code that accesses the device with assembly-language-level operations (*e.g.*, bit manipulations). For example, the C code needed to express the concatenation of the four lower bits of registers `y_high` and `y_low` is tedious. As shown in Figure 2-a, macros are often

defined so as to factorize common expressions or associate names with commands. Nevertheless, it is rather difficult to understand the behavior of the device from the implementation; maintenance of this code is error-prone and not easy.

Using Devil, driver development is a two stage process: first the chip is specified in Devil, then code is written using the stubs generated from the specification. *Describing* the device as opposed to *coding* improves readability. For instance, the Devil description of the variable `dy` in the Busmouse specification (see line 26 of Figure 1) consists of a straightforward concatenation of two bit-fragments. The Devil specification is so close to a device description that it can be used for documentation purposes.

When writing the driver code, the programmer first has to include Devil-generated stubs and to specify configuration information. For instance, in Figure 3-a, Busmouse stubs are used in debug mode and in a single device configuration (`#define DEVIL_NO_REF`). Further communication with the device is encapsulated in stubs (see Figure 3-b). Therefore, the driver programmer only has to focus on operating the device using abstract values. Writing the hardware operating code becomes a very simple task, especially if the programmer can use an existing Devil specification.

4.2 Robustness

As discussed in Section 3, Devil exposes properties that can be automatically checked. This section evaluates the benefits of these checks in terms of software robustness.

Detecting bugs as early as possible is crucial during the development process. A study by DeMillo and Mathur found that simple errors (*e.g.*, typographic errors, inattention errors) represent a significant fraction, though not the majority, of the errors in production programs. This study also revealed that such errors can remain hidden for a long time. Even though their study was concerned with the development of `TEX`, which differs from device drivers, these observations remain pertinent, and are even more important considering the permissive nature of a language such as C, es-

<pre>#define MSE_DATA_PORT 0x23c #define MSE_CONTROL_PORT 0x23e ... #define MSE_READ_Y_LOW 0xc0 #define MSE_READ_Y_HIGH 0xe0</pre>	<pre>dy = (inb(MSE_DATA_PORT) & 0xf); outb(MSE_READ_Y_HIGH, MSE_CONTROL_PORT); buttons = inb(MSE_DATA_PORT); dy = (buttons & 0xf) << 4; buttons = ((buttons >> 5) & 0x07);</pre>
<i>2a. Macro definition</i>	<i>2b. Macro usage</i>

Figure 2: Fragment of the original Linux driver for the Logitech Busmouse

<pre>#define DEVIL_NO_REF #define dev_name bm #define DEVIL_DEBUG #include "busmouse.dil.h"</pre>	<pre>bm_get_mouse_state(); dy = bm_get_dy(); buttons = bm_get_buttons();</pre>
<i>3a. Interface usage</i>	<i>3b. Stub usage</i>
<pre>#define bm_get_mouse_state() (\ outb(1, bm_cache.__dil_base__+2); bm_cache.cache_mouse_state.cache_get_x_high = inb(bm_cache.__dil_base__); \ outb(0, bm_cache.__dil_base__+2); bm_cache.cache_mouse_state.cache_get_x_low = inb(bm_cache.__dil_base__); \ outb(3, bm_cache.__dil_base__+2); bm_cache.cache_mouse_state.cache_get_y_high = inb(bm_cache.__dil_base__); \ outb(2, bm_cache.__dil_base__+2); bm_cache.cache_mouse_state.cache_get_y_low = inb(bm_cache.__dil_base__)) #define bm_get_dy() (\ (bm_cache.cache_mouse_state.cache_get_y_high & 0xfu) << 4 bm_cache.cache_mouse_state.cache_get_y_low & 0xfu) #define bm_get_buttons() ((bm_cache.cache_mouse_state.cache_get_y_high & 0xe0u) >> 5)</pre>	
<i>3c. Generated stubs (after inlining)</i>	

Figure 3: Fragment of the Devil based driver for the Logitech Busmouse

pecially when used to write low-level code.

In order to evaluate the impact of Devil on driver robustness, we have estimated the number of errors that can be detected automatically by the C and Devil compilers/checkers.³ The *error-detection coverage* is computed using a mutation analysis technique [1, 8].

For a program P , mutation analysis produces a set of alternate programs, each generated by modifying a single statement of P , according to mutation rules. In our experiment, the mutation rules introduce errors in operators, identifiers and literal constants. Such errors are generated by inserting, replacing or removing a character from the targeted token. For example, the logical operator `||` can be replaced by the bit operator `|`, the number 121 can be replaced by 21, etc. Mutation rules are defined so as to ensure that the resulting mutant is syntactically correct, and actually modifies the semantics of the program. Therefore, detection of the mutation introduced error by the com-

piler occurs only if the mutant violates a property of the language (*e.g.*, C or Devil).

In a C driver, we are only interested in testing the hardware operating code. Accordingly, we manually insert tags to mark the corresponding regions in the original C code, and only apply mutations to the tagged regions. In a Devil-based driver, mutations have to be applied both to the Devil specification of the device, and to procedure calls to the generated interface (this C code is denoted by C_{Devil} in the rest of the paper).

Our experiments compare the error-detection coverage of C against the error-detection coverages of the Devil specification and C_{Devil} . It should be noted that our measurements reflect the worst case for Devil for the following reasons. First, the mutation rules for C and Devil have been chosen so that C is always favored. Second, since a driver often uses a subset of a device, the Devil specification offers more mutation sites (possible errors) than the original C driver. Finally, Devil specifications should ideally come from the device manufacturer or widely

³In our current experiments, the benefit of run-time checks in Devil generated interfaces are not taken into account.

Device	Language lines		Number of mutation sites	Mutants per site	Undetected mutants per site	Mutation Sites with undetected mutants	Ratio to C
Logitech Busmouse	C	36	62	36.6	26.8	45.3	-
	Devil	21	81	15.9	0.2	1.0	-
	C _{Devil}	18	21	13.5	5.0	7.7	5.9
	Devil+C _{Devil}		102	15.4	1.2	8.7	5.2
IDE (Intel PIIX4)	C	64	95	29.0	18.8	61.8	-
	Devil	127	277	17.1	1.6	26.6	-
	C _{Devil}	81	42	22.6	7.4	13.3	4.6
	Devil+C _{Devil}		319	17.5	2.0	39.9	1.6
Ethernet (NE2000)	C	204	247	14.7	12.6	212.4	-
	Devil	144	456	15.0	1.1	33.7	-
	C _{Devil}	137	258	48.7	12.5	66.1	3.2
	Devil+C _{Devil}		714	27.2	4.7	99.8	2.1

Table 1: Language Error-Detection Coverage Analysis

available public-domain libraries. Thus, one can expect them to be bug-free and errors only to appear in C_{Devil}.

Measurement analysis. Our study focuses on three different devices (*e.g.*, Logitech Busmouse, NE2000 Ethernet, and IDE controllers) and their corresponding Linux 2.2-12 drivers. Table 1 presents the results of the mutation analysis. Overall, the experiments show that the probability of undetected errors is 1.6 to 5.2 times higher in C hand-crafted drivers than in Devil-based driver (Devil + C_{Devil}). When comparing C to C_{Devil} only (assuming that the specification is correct), the propensity of undetected errors 3.2 to 5.9 times higher in C. Finally, it can also be observed that mutation errors in Devil specifications are nearly always detected.

The first column of Table 1 represents the number of possible mutation sites (s). The second column shows the number of mutants (*i.e.*, errors) which can be injected for each site (m_s). For example, given an integer of two digits in base ten, 50 mutants can be generated (2 for removing a digit, 30 for inserting a new digit, and 18 for replacing a digit). The third column shows, for each mutation site, the number of mutants not detected by the compiler/checker (um_s).

To enable the comparison between C, Devil and C_{Devil} we are interested in measuring the number of mutation sites that have undetected mutants (s_{um}). To compute this value, we have to balance the number of undetected mutants per site by the number of mutation sites

($s_{um} = um_s / m_s * s$). For example, consider the Logitech Busmouse C driver. It has 62 mutation sites. For each site, 36.6 mutants are generated (on average) and 26.8 are not detected by the compiler. This give us 45.3 sites with undetected mutants.

4.3 Performance

It is well-recognized that the performance of drivers is critical for the overall system performance. Furthermore, as demonstrated by Thekkath and Levy for high-performance RPCs [18], the performance of the hardware operating code has a significant impact on the overall driver performance. While Devil can improve readability and robustness of driver hardware operating code, its usefulness depends on the efficiency of the generated code: using Devil must not induce significant execution overhead.

In order to evaluate the benefit and impact of Devil on driver development, we are re-engineering various Linux drivers and testing them on a bi-processor PC.⁴ Among the drivers and devices in a Unix system, we chose to implement first the IDE and the accelerated X11 drivers for two reasons: (i) they are representative of performance intensive drivers and they illustrate totally different device access behavior.

In the rest of this section, we first identify

⁴The PC is a DELL Precision 210 with the following configuration: two Pentium II 450 MHz, Intel PIIX4 PCI chipset, Maxtor model 91000D8 UDMA2 19.5Gb disk with 512Kb cache, 3Dlabs Permedia2 graphic controller.

Transfer mode	Sectors per interrupt	I/O Size in bits	Standard driver		Devil driver		Devil/Stand. throughput ratio
			I/O Operations	Throughput in Mb/s	I/O Operations	Throughput in Mb/s	
DMA	-	-	14	14.25	20	14.25	100 %
PIO	16	32	$7 + \frac{\#s(1+128)}{16}$	8.17	$10 + \frac{\#s(3+128)}{16}$	7.36	90 %
		16	$7 + \frac{\#s(1+256)}{16}$	4.45	$10 + \frac{\#s(3+256)}{16}$	3.94	88 %
	8	32	$7 + \frac{\#s(1+128)}{8}$	8.09	$10 + \frac{\#s(3+128)}{8}$	7.28	89 %
		16	$7 + \frac{\#s(1+256)}{8}$	4.42	$10 + \frac{\#s(3+256)}{8}$	3.91	88 %
	1	32	$7 + \#s(1 + 128)$	6.93	$10 + \#s(3 + 128)$	6.36	91 %
		16	$7 + \#s(1 + 256)$	4.06	$10 + \#s(3 + 256)$	3.63	89 %

Table 2: IDE Linux driver comparative performance results (using C loops)

the possible penalties induced by Devil, and then we compare the performance of the IDE and accelerated X11 Devil-based drivers with the original ones.

Micro-analysis Interface procedures generated by the Devil compiler contain I/O as well as bit-shift and bit-mask instructions. These procedures are optimized by the Devil compiler and implemented as pre-processor macros or inlined functions. Therefore, there is no execution overhead for a single Devil interface procedure as compared to hand-crafted C instructions.

In one situation, we observed that Devil could induce an execution penalty. Accessing independent device variables (*i.e.*, variables not grouped in a structure) defined over a single register, requires multiple Devil interface calls. Each additional call induces additional I/O, as compared to an hand-crafted driver. Nevertheless, as we found in our re-engineering of the IDE and Permedia2 driver, such variables are often parameters and rarely affect the performance of the critical path.

IDE driver Table 2 compares the performance of a Devil-based IDE driver with that of the original C driver. IDE throughput measurements were obtained using the standard Linux `hdparm` utility. We wrote two Devil specifications for this driver: a specification of the IDE controller and a specification of the Intel PIIX4 PCI busmaster IDE.

We have run the IDE driver in both Ultra

DMA-2 and several PIO modes, varying the size of I/O (16 or 32 bits) and the number of sectors transferred per interrupt. In DMA mode, Devil induces 6 additional I/O operations to prepare the command. Because of the long duration of the DMA transfer, there is no impact on the available throughput. In the PIO modes, there are 3 additional I/O operations to prepare the command, plus 2 for each interrupt ($\#s$ denotes the total number of sectors accessed). When using a C loop over a single variable read, we measured a 10% throughput penalty. When using block transfer stubs that use a `rep` instruction, we did not observe an impact on the available throughput.

Permedia2 X11 driver Tables 3 and 4 show the performance Devil-based X11 driver for the 3Dlabs Permedia2 graphics controller. Throughput measurements were obtained using the `xbench` utility. We have modified the 3Dlabs X11 server, which is based on a Xfree86-3.3.6 implementation. Although the Permedia2 chip provides acceleration for both 2D and 3D, the X11 server does not support 3D operations. Additionally, to minimize device-dependant code, many 2D primitives are implemented in software in Xfree86. In fact, hardware acceleration is only used for implementing the fill rectangle and screen area copy primitives.

Unlike many I/O devices, the Permedia2 controller maps registers into the memory address space. In fact, processor accesses are decoded by the controller and stored in a FIFO. Before accessing the chip, the driver must wait for free entries in the FIFO. This wait loop in-

Display Mode (bits/pixel)	Rectangle Size (pixels)	Standard Driver		Devil Driver		Devil/Stand. Throughput Ratio
		I/O Operations	Throughput (rect./s)	I/O Operations	Throughput (rect./s)	
8	2x2	$3(\#w) + 15$	984838	$3(\#w) + 17$	949052	96 %
	10x10		585621		585350	99 %
	100x100		38472		38438	100 %
	400x400		3762		3762	100 %
16	2x2	$3(\#w) + 15$	982338	$3(\#w) + 17$	945916	96 %
	10x10		333670		332499	100 %
	100x100		21022		21033	100 %
	400x400		2221		2221	100 %
24	2x2	$2(\#w) + 10$	978605	$2(\#w) + 10$	945884	97 %
	10x10		235119		234716	100 %
	100x100		3693		3693	100 %
	400x400		244		243	100 %
32	2x2	$3(\#w) + 15$	957534	$3(\#w) + 17$	929833	97 %
	10x10		251522		251584	100 %
	100x100		10466		10466	100 %
	400x400		899		899	100 %

Table 3: Comparative Performance of Permedia2 Xfree86 Driver: Rectangle Test

Display Mode (bits/pixel)	Copy Size (pixels)	Standard Driver		Devil Driver		Devil/Stand. Throughput Ratio
		I/O Operations	Throughput (copies/s)	I/O Operations	Throughput (copies/s)	
8	2x2	$3(\#w) + 15$	149553	$3(\#w) + 17$	144494	97 %
	10x10		123584		122300	99 %
	100x100		10662		10638	100 %
	400x400		764		764	100 %
16	2x2	$3(\#w) + 15$	145084	$3(\#w) + 17$	136755	94 %
	10x10		85994		85561	99 %
	100x100		3502		3512	100 %
	400x400		238		238	100 %
24	2x2	$2(\#w) + 9$	144385	$2(\#w) + 9$	144521	100 %
	10x10		77443		77605	100 %
	100x100		1716		1716	100 %
	400x400		114		114	100 %
32	2x2	$2(\#w) + 9$	142335	$2(\#w) + 9$	142598	100 %
	10x10		69762		69804	100 %
	100x100		1703		1701	100 %
	400x400		111		111	100 %

Table 4: Comparative Performance of Permedia2 Xfree86 Driver: Screen Copy Test

duces one I/O operation per iteration. In Tables 3 and 4, $\#w$ denotes the number of iterations per wait loop. In the driver we modified, 2 or 3 wait loops are performed per primitive call.

The time for execution of a drawing command by the Permedia2 controller is proportional to the number of drawn pixels and their depth. Therefore, the overhead induced by Devil is more perceptible for shortest commands. The worst case is reached for 2x2 pixel commands in 8 or 16 bit mode, where Devil induces a performance penalty of up to 6%. For primitive calls involving more than 100 pixels (which are the most common in practice), 99% to 100% of the performance of the original server is obtained (always 100% in 24 bit mode).

5 Related Work

Our work on device drivers started with a study of graphic display adaptors for a X11 server. We developed a language, called GAL, aimed at specifying device drivers in this context [19]. Although successful as a proof of concept, GAL covered a very restricted domain.

The goal of the UDI project⁵ is to make device drivers source-portable across OS platforms. To do so, they have normalized the API between the OS and the lower part of device drivers [14]. Besides showing the timeliness of our work, UDI focuses only on the high-level

⁵The UDI (Uniform Driver Interface) project is the result of a collaboration of several computer companies including Compaq, HP and IBM.

part of drivers and their interaction with the OS.

Windows-specific driver generators like BlueWater System's WinDK [4] and NuMega's DriverWorks [6] provide a graphical interface for specifying the main features of a driver. They produce a driver skeleton that consists of invocations of coarse-grained library functions. To our knowledge, no existing driver generators cover the communication with the device.

Languages for specifying digital circuits and systems have existed for many years. The VHDL standard [11], widely used in this domain, is one of the most expressive. It addresses several aspects of chip design such as documentation, simulation and synthesis. VHDL provides both high-level and low-level abstractions: arrays and loops are supported, as well as bit-vector literals and bit extraction. However, all VHDL abstractions focus on the inner workings of circuits, not their high-level programming interface. As a consequence, chip interfaces are not explicitly denoted, and VHDL compilers perform limited consistency checks. Interestingly, VHDL allows attaching arbitrary strings to variables. Using them to add interface-specific information is possible, but would require a normalized syntax and compiler support, which in some way amounts to embedding Devil concepts in VHDL.

The New Jersey Machine-Code Toolkit [15] helps programmers write applications that process machine code at an assembly-language level of abstraction. Guided by a instruction set specification, the toolkit generates the code for reading or generating binary. Some simple verifications are also done at the specification level.

6 Conclusion and Future Work

This paper has presented a new approach to developing hardware operating code that is based on an IDL named Devil. This IDL enables hardware communication to be described using high-level, domain-specific constructs instead of being written with assembly-language-like operations. Raising the implementation level of this layer of a device driver dramati-

cally reduces the risk of errors. Devil has shown to be expressive enough to specify a wide variety of devices such as the DMA, interrupt, Ethernet, IDE disk, sound, mouse and video controllers.

Because Devil significantly raises the level of abstraction of communication with the hardware, Devil specifications are more readable, maintainable and re-usable than equivalent C code.

We have developed a compiler that checks the consistency of a Devil specification and automatically generates low-level code that is mostly comparable to hand-crafted code. We have assessed our approach by conducting experiments aimed at comparing hardware operating code in C or Devil for robustness and performance. We have demonstrated that our approach enables hardware operating code to be more robust than C, with mostly comparable performance.

Our future work aims to improve the performance of the output of our Devil compiler. Specifically, we want to enhance performance by factorizing and scheduling device communications and by better exploiting special-purpose assembly-level instructions. The key advantage of introducing optimizations at the compiler level is that these advanced techniques are transparently available to any Devil programmer. As a result, our work reduces the need to have a highly experienced programmer to write hardware operating code since part of this expertise is captured by the compiler.

We are currently building a public domain library of Devil specifications for common devices such as those found in PCs. Our purpose is to setup a WWW repository that would help dissemination of expertise about hardware and facilitate the development of device drivers.

Acknowledgment.

We thank Julia Lawall from DIKU and the other members of the Compose group for helpful comments on earlier versions of this paper. We also thank Timothy Roscoe and the anonymous reviewers for their valuable inputs.

This work has been partly supported by France Telecom under the CTI contract 991B726, the French Ministry of Research and Technology under the Phenix contract 99S0362, and the French Ministry of Education and Research.

Availability

The Devil compiler, Devil specifications and Devil-based drivers mentioned in the paper are available at the following web page <http://www.irisa.fr/compose/devil>.

References

- [1] A. T. Agree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation analysis. Technical Report GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, USA, September 1979.
- [2] B.N. Bershad, T.E. Anderson, E.D. Lazowska, and H.M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.
- [3] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.
- [4] BlueWater Systems, Inc. *WinDK Users Manual*. URL: www.bluewatersystems.com.
- [5] Cirrus Logic, Inc, P.O. Box 17847, Austin, TX 78760. *CrystalClear™ Single Chip Audio System (CS4236B)*, September 1997. URL: www.cirrus.com.
- [6] Compuware NuMega. *DriverWorks User's Guide*. URL: www.numega.com.
- [7] E. N. Dekker and J. M. Newcomer. *Developing Windows NT device drivers : A programmer's handbook*. Addison-Wesley, first edition, March 1999.
- [8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.
- [9] R. Draves, M. Jones, and M. Thompson. *MIG - The MACH Interface Generator*. School of Computer Science, Carnegie Mellon University, July 1989.
- [10] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom. Flick: A flexible, optimizing IDL compiler. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 44–56, Las Vegas, NV, USA, June 15–18, 1997.
- [11] IEEE Standards. *1076-1993 Standard VHDL Language Reference Manual*, 1994. URL: standards.ieee.org.
- [12] H. P. Messmer. *The Indispensable PC Hardware Book*. Addison-Wesley, third edition, 1997. page 669, figure 26.6.
- [13] S. O'Malley, T. Proebsting, and A.B. Montz. USC: A universal stub compiler. In *Proceedings of Conference on Communication Architectures, Protocols and Applications*, London (UK), September 1994.
- [14] Project UDI. *UDI Specifications, Version 1.0*, September 1999. URL: www.project-udi.org.
- [15] Norman Ramsey and Mary F. Fernandez. The new jersey machine-code toolkit. In *Proceedings of the Winter USENIX Conference*, New Orleans, LA, January 1995.
- [16] A. Rubini. *Linux Device Drivers*. O'Reilly, first edition, February 1998.
- [17] L. Réveillère, F. Méryllon, C. Consel, R. Marlet, and G. Muller. The Devil language. Research Report 1319, IRISA, Rennes, France, May 2000.
- [18] C.A. Thekkath and H.M. Levy. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.
- [19] S. Thibault, R. Marlet, and C. Consel. Domain-specific languages: from design to implementation – application to video device drivers generation. *IEEE Transactions on Software Engineering*, 25(3):363–377, May–June 1999.

Taming the Memory Hogs: Using Compiler-Inserted Releases to Manage Physical Memory Intelligently

Angela Demke Brown and Todd C. Mowry

School of Computer Science

Carnegie Mellon University

Pittsburgh, PA 15213

{demke, tcm}@cs.cmu.edu

Abstract

Out-of-core applications consume physical resources at a rapid rate, causing interactive applications sharing the same machine to exhibit poor response times. This behavior is the result of default resource management strategies in the OS that are inappropriate for memory-intensive applications. Using an approach that integrates compiler analysis with simple OS support and a run-time layer that adapts to dynamic conditions, we have shown that the impact of out-of-core applications on interactive ones can be greatly mitigated. A combination of prefetching pages that will soon be needed, and releasing pages no longer in use results in good throughput for the out-of-core task and good response time for the interactive one. Each class of application performs well according to the metric most important to it. In addition, the OS does not need to attempt to identify these application classes, or modify its default resource management policies in any way. We also observe that when an out-of-core application releases pages, it both improves the response time of interactive tasks, and also improves its own performance through better replacement decisions and reduced memory management overhead.

1 Introduction

Many of the computational problems of interest to scientists and engineers involve data sets that are much larger than physical memory [6, 7, 17]. Despite the continuing trend toward larger memories, it is unlikely that these data sets will ever fit entirely within main memory. Increases in processor power and memory capacity make it feasible to solve larger problems, or to solve the same problem at a finer granularity, but the size of the data set grows with the problem being solved. For instance, input data sets for scientific visualization can currently exceed 100 Gbytes [5]. For these “out-of-core” applications, I/O is required throughout the execution of the program to bring data into memory as it is needed and possibly to move it back out to disk. Performance concerns have traditionally forced programmers to explicitly manage the I/O in their out-of-core codes. Recently, however, we demonstrated that paged virtual memory can be enhanced with prefetching to effectively hide the latency of page faults without

placing any burden on the programmer [15]. In this approach, the compiler provides information on future access patterns, the OS supports a simple *prefetch/release* interface, and a run-time layer improves performance by adapting to dynamic behavior.

While this earlier work demonstrated that out-of-core applications can achieve excellent performance on a dedicated machine, it would be far more cost-effective if these tasks could coexist with other applications in a multiprogrammed environment. Unfortunately, out-of-core tasks have the potential to severely degrade the performance of other tasks which are attempting to use the machine at the same time. This problem arises because operating on massive data sets consumes physical resources (memory and disk bandwidth) at a rapid rate, displacing the working sets of other applications and increasing their page fault service times. To make matters worse, successful prefetching causes physical resources to be consumed even faster, increasing the negative impact on other applications.

1.1 Impact on Interactive Performance

In many cases, the excessive resource consumption by out-of-core tasks is caused not by inherent resource requirements, but rather by sub-optimal resource management policies in the OS. While the default policies perform well in most cases, they are poorly suited to the demands of memory-intensive programs. For instance, most commercial operating systems use a global page replacement algorithm, which allows pages to be stolen from any application to satisfy page faults. Interactive tasks are particularly vulnerable in such an environment since they are unable to defend their memory effectively. Consider an editor program which may have no memory system activity for several seconds while it waits for user input. A program computing the inner product of two out-of-core vectors could easily sweep through all of physical memory in this time, stealing pages from the editor as they move to the head of the LRU queue. In this case, the out-of-core computation could have achieved the same performance using only two pages of physical memory, allowing the editor to remain responsive regardless of the intervening delay.

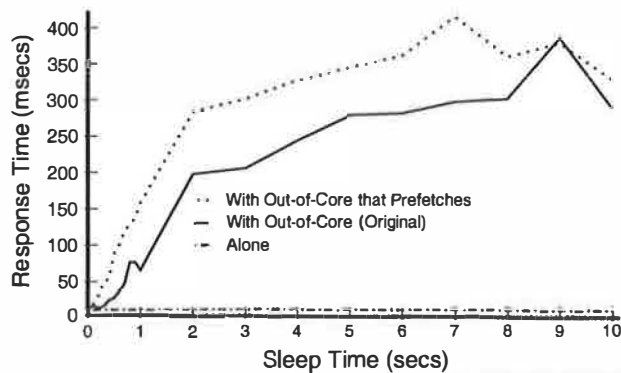


Figure 1. Impact of sharing the machine with an out-of-core matrix-vector multiplication (MATVEC) on the response time of an interactive task across a range of sleep times between touching 1 MB of data.

To illustrate the impact of out-of-core applications on interactive performance, we ran the following experiment on a 4-processor SGI Origin 200 configured to have approximately 75 MB of memory available to user programs.¹ A simple program emulates the memory system behavior of an interactive task by repeatedly touching a 1 MB data set, then sleeping for a fixed amount of time. By varying the amount of sleep time we can control the frequency with which each page of the “interactive” task is accessed. The “response time” is the time to touch the entire data set. This program is run concurrently with one that repeatedly performs a matrix-vector multiplication on an out-of-core data set (400 MB). The results are shown in Figure 1. With no sleep time, the “interactive” task defends its memory extremely well, achieving the same response time as on a dedicated machine. As the sleep time increases, however, the task incurs an increasing number of page faults and the response time rises. When the out-of-core program uses prefetching, the response time begins to increase at much shorter sleep times, grows much faster, and rises to a higher level. Prefetching combined with global replacement puts the interactive task at a serious disadvantage.

In recognition of the shortcomings of existing OS policies, a significant amount of recent research has focused on customizable operating systems. While a customizable OS could provide the flexibility to tailor the resource management policies for out-of-core codes, our results in this paper demonstrate that we can achieve the desired outcome (i.e. customizable behavior) in this particular case through relatively modest extensions of today’s commercial operating systems. To accomplish this goal, we adopt a strategy similar to our earlier work [15] in which the OS, compiler, and a run-time layer all cooperate. The role of the OS is to perform global resource allocation across all applications while the role of each out-of-core application (via the com-

¹This amount of memory is artificially low for modern systems, but makes it possible to run experiments on out-of-core programs in a reasonable amount of time. Similar behavior can be seen with more memory and larger out-of-core programs, although the time required to consume all physical memory increases with the amount of memory available

piler and run-time layer) is to effectively manage the resources it has been granted.

1.2 Objectives of This Study

In our earlier study [15], our focus was using *prefetching* to hide the *I/O latency* of out-of-core applications running on a *dedicated* machine. In this study, we focus on using *release* operations to *manage physical memory* intelligently within a *multiprogramming workload* that includes an out-of-core application. Although we introduced the concept of release operations in that earlier paper, we made little use of them because they offered no significant performance benefit to stand-alone out-of-core applications on the research prototype OS (Hurricane [21]) and machine (Hector [22]) that we used. Note that we observe a different result in this study using a modern commercial OS and machine.

The primary contribution of this paper is that we propose, implement, and evaluate a solution to the problem of preventing out-of-core applications from ruining the response time of interactive applications while still enjoying the performance benefits of aggressive I/O prefetching. Our solution uses the compiler to automatically insert *release* hints (in addition to *prefetch* hints) into the out-of-core application while a run-time layer and OS provide appropriate support. This approach requires minimal changes to existing operating systems and places no additional burden on the programmer. We implement our solution within a modern commercial system (an SGI Origin 200 running our modified version of IRIX 6.5) and evaluate its performance impact on both out-of-core applications and interactive tasks sharing the same machine.

The remainder of this paper is organized as follows. Section 2 motivates allowing applications to manage their own resources, and describes the features we feel are needed to do so effectively. Section 3 describes the components of our system and their implementations. Section 4 presents our experimental results, and we discuss related work and draw conclusions in Sections 5 and 6.

2 Memory Management Strategies

The goal of a virtual memory management system in a multiprocessor environment is to share the physical memory resources among all the competing applications. Most operating systems provide policies that perform well in the common case, but exhibit bad behavior when a memory-intensive program is sharing the machine with others. In this section we discuss why it may be beneficial to give demanding applications control over their own memory management, and examine some forms such control could take. Finally, we outline the features we believe are necessary for an effective system that allows applications to explicitly manage their memory resources.

2.1 Global vs. Local Replacement

An out-of-core task can degrade the responsiveness of an interactive task because global replacement policies select victims from among all the pages in the system with-

out regard to ownership. In contrast, a local page replacement strategy helps to isolate each process from the paging activity of others. Each process is allocated a fixed set of physical pages and a victim is selected from among them as needed. Thus, interactive tasks would not have to worry about losing pages to a demanding out-of-core program. Unfortunately, poor memory utilization may occur, as pages are not allocated to processes according to their need. Attempting to determine the right number of pages to allocate to each process and dynamically adjusting this number during execution can improve memory usage but greatly complicates the OS. In practice, most workstation operating systems use global page replacement.

Although local replacement policies can insulate processes from each other, they may not provide the best replacement policy for each application. Rather than altering the overall strategy employed by the OS, it is preferable to modify individual applications so that their competition for physical resources better reflects their actual needs. This approach enables applications to improve their own performance through local replacement decisions that are superior to those used by the OS. The largest drawback of specializing applications to do memory management is the burden placed on the programmer; however, we propose a framework in which all the necessary modifications are performed automatically by a compiler.

2.2 Application-Managed Replacement

Giving specialized applications more control over their own memory management to improve their performance has been suggested before. For instance, the Mach OS supports external pagers to allow applications to control the backing storage of their memory objects [18]. Extensions to the external pager interface have been used to implement user-level page replacement policies [14], and to support discardable pages (i.e. dirty pages that do not need to be written to backing store) [20]. In contrast, our approach shows that specialized applications can and should exploit extra control for the benefit of other applications executing concurrently. This is especially true for programs that use prefetching to improve their own performance since the gains they enjoy impose a heavy penalty on other processes sharing the system. In this case, the OS could require that prefetching applications also explicitly release pages.

Given that application-controlled memory management is desirable, one possibility is for the OS to allow applications to choose from a small set of "reasonable" replacement policies. This strategy does not require much effort on the part of the application programmer, but also does not provide a great deal of power or flexibility. Another possibility is for the OS to provide a more general interface that allows applications to explicitly specify which of their pages can be reclaimed. This approach is preferable since individual applications can implement a variety of replacement policies tailored to their specific needs.

Application management of memory resources through an interface that allows individual pages to be specified can

be either *reactive* or *pro-active*. In a *reactive* approach, the OS notifies the application when one or more of its pages is about to be reclaimed. The application can then implement its own replacement policy by telling the system which pages to take. This is essentially the approach taken by the VINO page eviction extension [19], for example. A reactive system benefits applications that can make better replacement decisions than the default OS policy, and has the advantage of delaying the decision until memory actually needs to be reclaimed. Unfortunately, it will not help isolate other applications from a memory-intensive one—the OS still decides which processes should give up pages.

In a *pro-active* system, an application returns pages to the system *before* they are strictly required, either as soon as they are no longer needed or based on some other criteria such as the amount of free memory. A pro-active approach can obviate the need for the OS to steal pages by increasing the global pool of free memory, thus providing benefit to all applications sharing the system. Of course, the pro-active approach is not without potential cost to the application using it. If the decision to release memory is made without full knowledge of future accesses, as is typically the case, then the application may give up pages that are still useful.

Our goal is to develop a system that allows applications to pro-actively return memory to the system on a page-by-page basis, to the mutual benefit of themselves and other concurrently executing applications without placing any additional burden on the programmer. We now outline the elements that we believe are necessary to achieve this goal.

2.3 Requirements for Effective Application-Directed Memory Management

If applications are to manage their own memory usage, the first requirement is some form of support from the OS for this type of activity. Second, to automate memory management without rewriting the application source code, we will need compiler analysis to detect access patterns and insert the necessary paging operations. Finally, since good replacement decisions will depend on dynamic conditions during program execution, we will need a run-time layer to intercept the information provided by the compiler and adapt the application's behavior as required.

2.3.1 Operating System Support

The OS must supply both primitive operations and additional information to applications. The operations should allow the application to specify the virtual memory addresses that it will need in the future as well as those that it no longer needs. The additional information is needed to allow the application to make informed decisions about when memory management activity is required. It should include information about which virtual pages are currently in memory, how many pages are currently in use, and the upper limit on pages that the application should use.

2.3.2 Compiler and Run-time Support

To determine whether a given page should be released at a particular point, the compiler attempts to answer the following questions. First, will the page be referenced again

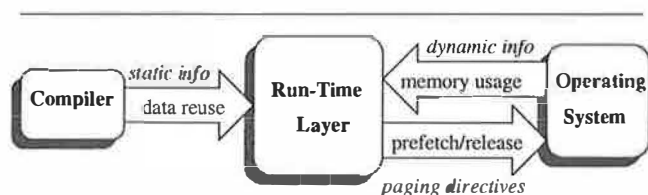


Figure 2. Information flow between components of our system.

in the future? If not, then a release hint is inserted. Second, does the number of other unique pages that will be accessed before the page is reused exceed the expected amount of available memory? If so, then the page is unlikely to remain in memory, and a release hint is inserted. Otherwise, release hints are not inserted.

There is a certain duality between the analysis for inserting prefetches and releases. In both cases, the compiler attempts to model when pages are being reused, and whether enough intervening accesses exist between these reuses to cause displacement. For prefetching, the question is whether a given page *has* remained in memory since its *last* reuse (if so, we do not need to insert a prefetch hint for it); for releasing, the question is whether a given page *will* remain in memory until its *next* reuse (in which case we do not want to release it). One difference, however, is that prefetching uses this analysis only to minimize overheads—the latency-hiding benefit of prefetching depends only on scheduling prefetches early enough—whereas the benefit of release hints depends directly on the quality of this reuse analysis.

Ideally, the compiler would be able to analyze the data accesses perfectly and insert these paging directives precisely where they are needed. However, this ideal is not realistic for the following two reasons. First, one cannot always predict memory access patterns with only static information. They may depend on run-time parameters (such as the problem size for the current run) or be data-dependent (such as the indirect references that often occur in sparse-matrix programs, e.g., `a[b[i]]`). While it is possible to issue prefetches for indirect references [8, 15], it is not possible to reason statically about any reuse that they may have, and hence it is not clear that the compiler can generate useful release hints for them. The second major limitation of the compiler is that it decides when reuse can be exploited based on an assumption of how much memory will be available to the application at run-time. In a multiprogrammed environment, such assumptions may be wildly inaccurate, especially since the amount of available memory may fluctuate dynamically during execution.

For these reasons, it may be undesirable to actually release a page at the point where the compiler has inserted the corresponding release hint. Instead, a run-time layer should collect information about pages that could be released, according to the compiler-generated addresses, and actually perform the releases only when necessary. In addition to the addresses of releasable pages, the compiler should include some indication of whether it believes the released pages will be used again or not. The role of the run-time

layer is to use the information provided by the OS and the compiler to answer the following questions: When should memory be returned to the OS? How many pages should be released? Which of the “releasable” pages should actually be given up? Figure 2 depicts the flow of information from the compiler and the OS to the run-time layer.

The decision of when to release memory depends primarily on how close the application is to the upper limit on memory usage suggested by the operating system. The decision of how much memory to release is more complicated. The run-time layer needs to balance the desire to remain below the OS limit, the desire to retain as much memory as possible, and the desire to perform release operations as infrequently as possible to minimize overhead. For example, suppose the run-time layer detects that the application is close to its upper memory limit, and has knowledge of 1000 pages that could be released. By releasing all of these pages, the run-time layer increases the amount of time before it will have to act again, but it may have given up pages that would be used again in the future by acting too aggressively. The run-time layer should also consider the application’s expected future need for memory when deciding how much to release. If the application is close to the upper memory limit, but only needs a small number of additional pages, the run-time layer may not need to release memory at all. Finally, once the run-time layer has determined that a release is necessary, and has decided how many pages to release, it must choose which pages should actually be returned to the OS. This decision depends on the expected future use of these pages; the run-time layer’s choice should be guided by information from the compiler.

There are two situations that may arise from the compiler analysis. First, the compiler may have inserted release hints because it has determined that the page will not be reused again. The run-time layer should release these pages before any pages that are known to have reuse. Second, the compiler may have detected that data reuse existed, but inserted release hints anyway because the volume of data accessed between reuses was expected to flush the page from memory. For these pages, the run-time layer should perform releases according to the intrinsic data reuse (which can be revealed by the compiler), attempting to keep as much data in memory as possible for the subsequent accesses. For instance, suppose the application is repeatedly accessing an array that is much larger than physical memory. The run-time layer can implement *most recently used* (MRU) replacement once the memory usage approaches the upper limit set by the OS, thus keeping at least the first portion of the array in memory for future use.

2.4 An Example

To help illustrate these concepts, we now present a simple example. Figure 3(a) shows the source code for a calculation that averages an element of a matrix with its neighbors, while Figure 3(b) depicts the data elements that are touched during a single iteration of the innermost loop. The references have temporal reuse along the *i* dimension

(a) Source code for averaging nearest-neighbors

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    a[i][j] = (a[i+1][j-1] + a[i+1][j]
              + a[i+1][j+1] + a[i][j-1] + a[i][j]
              + a[i][j+1] + a[i-1][j-1] + a[i-1][j]
              + a[i-1][j+1])/9.0;
```

(b) View of data references to the matrix a

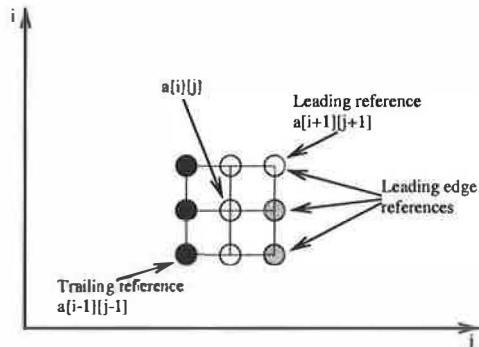


Figure 3. Example source code showing multiple references with different types of reuse, and graphical view of the data accesses during a single iteration of the innermost loop.

(since the items accessed at $a[i+1][*]$ are touched again in the next iterations of the i -loop). There is spatial reuse along the j dimension, and there may also be spatial reuse along the i dimension, depending on the length of the rows.

We can identify two major working sets in this access pattern. At the smallest level, we need to hold the leading edge of the data access square (those references indexed by $j+1$) in memory, requiring at most one page for each of the three references on this edge. Except at page boundaries, the references indexed by $j-1$ will fall on the same page as this leading edge due to spatial reuse. We therefore need at most six pages to fully exploit the spatial reuse along the j dimension. The second level working set exploits the temporal reuse along the i dimension, requiring us to hold three rows of the matrix in memory, so that the row first indexed by $i+1$ in one iteration will still be available for the i and $i-1$ references in the subsequent iterations. Of course, there is also a third level, which corresponds to keeping the entire matrix in memory.

The compiler can determine precisely which references to prefetch and release if it has the dimensions of the matrix and a good estimate of the physical memory available. To successfully exploit the reuse across iterations of the i loop, we need to retain three rows of the matrix in memory. If this is possible, then a prefetch will be inserted only for the leading reference, $a[i+1][j+1]$, and a release will be inserted for the trailing reference, $a[i-1][j-1]$. This corresponds to keeping the second level working set in memory. If the amount of memory needed to hold three rows is less than the amount available, the compiler will instead decide to prefetch all three references on the leading edge of the data access square (i.e. the $a[i+1][*]$ references) and release the references on the trailing edge, corresponding to the first level working set. If the dimensions of

the matrix are unknown at compile-time, the compiler must choose between these two options. Since over-estimating the ability of memory to retain data leads to missed opportunities (both for prefetching and releasing), it is preferable to assume that only the smallest working set will fit in memory. The run-time layer is responsible for reducing the overhead of unnecessary operations that result.

Having outlined the features that we believe are necessary to achieve a good pro-active user-level memory management system, we turn now to a discussion of the specific components in our prototype system.

3 Overview of Prototype System

Our prototype system consists of three major components: extensions to the OS, a compiler analysis pass, and a run-time layer. We now describe these components.

3.1 Implementation of OS Support

We have implemented support for user-level paging directives (i.e. prefetch and release) within the SGI IRIX 6.5 operating system. IRIX 6.5 supports *policy modules* (PMs) that allow users to select various memory management policies for page size, allocation, migration, and replication. A PM may be connected to any range of an application's virtual address space, down to the level of a single page. We have defined a new PM—called “PagingDirected”—that allows a user-level process to invoke prefetch and release operations on pages of its address space. In addition, the PagingDirected PM shares information about memory usage with the application through a single 16KB page.

3.1.1 Managing the Shared Page

The shared page is allocated by the OS and mapped read-only into the application's address space when the PagingDirected PM is created. The page is used primarily as a bitmap, indexed by virtual page number, in which bits are set to indicate that the corresponding page is in memory, and cleared otherwise. The first two words in the page are reserved, however, to indicate the current number of pages in use by the process, and the upper limit on pages that the process should be using, respectively.

All updates to the shared page are handled by the OS. When the PagingDirected PM is created, all bits in the shared page are initially set. When the application attaches the PM to a region of its virtual address space, the bits corresponding to those addresses are all cleared. Thereafter, bits are set whenever a physical page is allocated for a virtual page associated with this PM, either due to prefetch requests or ordinary page faults. Bits are cleared when pages are reclaimed, either by an explicit release request or due to default page replacement activity. The estimates of current and maximum usage are updated only when the process experiences some type of memory system activity, rather than every time the information changes. One consequence of this approach is that an application's upper limit may drop dramatically if another process begins using memory (reducing the total free memory in the system),

but the first process will not be informed of this change until it issues a prefetch/release request, page faults, or has memory stolen from it. The alternative approach of immediate updates would require the OS to either maintain a list of processes that should be informed, or to scan the list of all processes each time the amount of free memory in the system changes. This additional expense does not appear to be justified. Another alternative that we have not explored would be to notify interested applications if conditions change by more than a set threshold, rather than waiting for memory activity to occur.

3.1.2 Handling Prefetch and Release Requests

When the PagingDirected PM receives a request to prefetch a page, it performs actions similar to those that occur for a page fault, with two notable exceptions. First, if there is no free memory, the request is discarded immediately. This feature prevents memory from being stolen to satisfy prefetches when the demand for memory is high. Second, when the request completes, the prefetched page is not fully validated and no entry is made in the TLB. This feature prevents mappings for prefetched pages from displacing TLB entries which are still in use.

Requests to release pages are handled by passing the addresses to a new system releasing daemon—called the *releaser*—which functions similarly to the paging daemon, but is specialized to reclaim only the pages indicated by the application. When a release request is made, the PagingDirected PM clears the bits for the pages and enters the request in the releaser's work queue. The releaser handles requests as they are received, first checking the bit vector to make sure that the pages have not been referenced again (either by a prefetch or a real reference) since the time of the request. The releaser then performs all actions needed to free the pages, including writing back dirty pages. Released pages are placed at the end of the free list, giving pages that were released too early a chance to be rescued.

3.1.3 Setting the Memory Limit

The goal in setting the upper limit on memory usage is to prevent the default page replacement policies from being activated, if at all possible. IRIX provides a number of tunable system parameters that control when pages will be stolen; these parameters can be also used by the PagingDirected PM in an effort to prevent such activity. First, the maximum number of pages that any process can have resident in memory (*max_rss*) can be set. If a process exceeds this limit, the system paging daemon will attempt to trim physical pages from it. Second, the minimum number of pages that should be kept free (*min_freemem*) can be set. If total free memory falls below this limit, the paging daemon will steal pages from all processes in the system according to an approximation of an LRU policy.

If physical memory is ample, it is sufficient to tell the process to remain below *max_rss*. When memory is limited, the process should be encouraged to use no more than its current memory usage (*current_size*), plus the amount of free memory in the system (*tot_freemem*), less

min_freemem. The recommended upper limit on memory usage in our system is thus given as follows:

$$\text{upper limit} = \min(\text{max_rss}, (\text{current_size} + \text{tot_freemem} - \text{min_freemem})) \quad (1)$$

Note that in setting this upper limit we are not guaranteeing that the application will be able to allocate this many pages for itself. Instead, the upper limit is an indication of the number of pages for which the application is allowed to compete. Pages that have already been allocated to another process are not part of the global free memory pool and thus may not be acquired by the prefetching application. One result of this decision is that the upper memory limit is a moving target which is dynamically adjusted as the total demand for physical memory by all applications changes. Thus, the OS does not try to determine the “right” amount of memory to allocate to each process, it simply tells interested processes how much memory is still available. Finding the right amount of memory for each process is beyond the scope of this paper.

3.2 Implementation of Compiler Analysis

We implemented our compiler algorithm as a pass in the SUIF (Stanford University Intermediate Format) compiler [9]. This algorithm is an extension of the algorithm we developed earlier for inserting prefetching hints into array-based codes [15]; pointer-based data structures are not currently handled, although techniques used for cache prefetching may be applicable [13]. We now briefly describe our algorithm. The following parameters are given to the compiler to describe the target system: the size of main memory, the page size, and the page fault latency. The compiler first uses *reuse analysis* to detect the intrinsic data reuses in the access patterns, then uses the page size and memory size parameters to apply *locality analysis* to predict when misses (i.e. page faults) are likely to occur. References that are likely to suffer page faults are isolated through *loop splitting* techniques, and prefetches for these references are scheduled based on the latency parameter using *software pipelining*. Figure 4 shows the process of creating the specialized executable from the original source code. The compiler analyzes each set of nested loops independently, thus reuses that occur between independent sets of loops are not considered. While the earlier algorithm did insert release hints in some cases, we have extended that analysis in two major ways: (i) we insert releases far more aggressively, and (ii) we encode reuse information into the release hints to allow the runtime layer to choose which pages to release first.

Given the existing locality analysis, it is relatively straightforward to generate release operations. During locality analysis, the compiler identifies groups of references that effectively share the same data and can be treated as a single reference—this is called “group locality”. For each of these groups (a group may contain only a single reference), the compiler identifies the *leading reference*

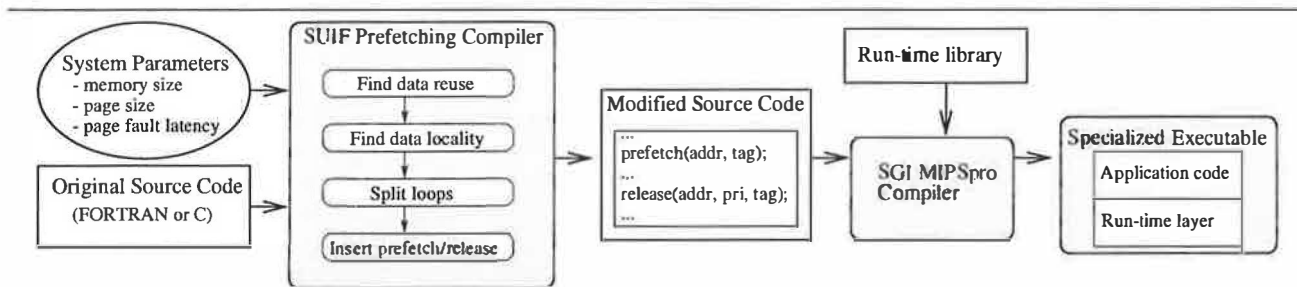


Figure 4. Steps in the automatic transformation of original application into prefetching/releasing executable.

(i.e. the first reference to access the data) as the reference to prefetch—we simply extend this analysis to also identify the *trailing reference* (the last one to touch the data) as the address to release. For indirect references (e.g., `a[b[i]]`), we do not insert a release request since it is too hard to predict whether the data will be accessed again.

In addition to identifying the addresses of data that can be released, the compiler also indicates whether the data has temporal reuse, and how soon the reuse is expected, based on the reuse analysis. (Recall that releases may be generated because the reuse is not expected to result in locality). The reuse information is encoded as a priority value which is passed as a parameter in the release requests; larger numbers represent references with earlier reuse—i.e. those which we would most prefer to retain in memory. The release priority is calculated as follows. Let $depth(i)$ denote the depth of loop i , with the outermost loop nest having a depth of 0. Let $temporal(x)$ be the set of nested loops in which reference x has temporal reuse. The release priority is computed by the following equation:

$$priority(x) = \sum_{i \in temporal(x)} 2^{depth(i)} \quad (2)$$

The run-time layer can use this information to prioritize which pages are actually returned to the system when the memory usage approaches the upper limit, attempting to retain those pages that will be reused earlier to reduce the total amount of paging.

Figure 5 shows an example of the output of our compiler for a set of loops that repeatedly perform a matrix-vector multiplication. The compiler analysis has determined that references to the `b` array have temporal reuse with respect to both the `i`-loop and the `iter`-loop, but that this reuse is not expected to result in locality since the volume of data accessed between reuses is more than the memory size parameter. In contrast, references to the `a` array have temporal locality with respect to the `iter`-loop only. Both array references have spatial reuse (and locality) causing the compiler to schedule prefetches for the first reference to each page, and releases after the last reference to each page. Using equation (2), a release priority of 1 is assigned to the releases for the `a` array, and a priority of 3 is assigned to the releases for the `b` array, indicating that `b`'s pages will be reused before `a`'s pages. Neither prefetches nor releases are inserted for the `c` array since this item is smaller than a page and is expected to remain in memory.

(a) Original Code

```
int a[100][1000000];
int b[1000000];
int c[100];

for (iter = 0; iter < 10; iter++)
  for (i = 0; i < 100; i++)
    for (j = 0; j < 1000000; j++)
      c[i] = c[i] + a[i][j]*b[j];
```

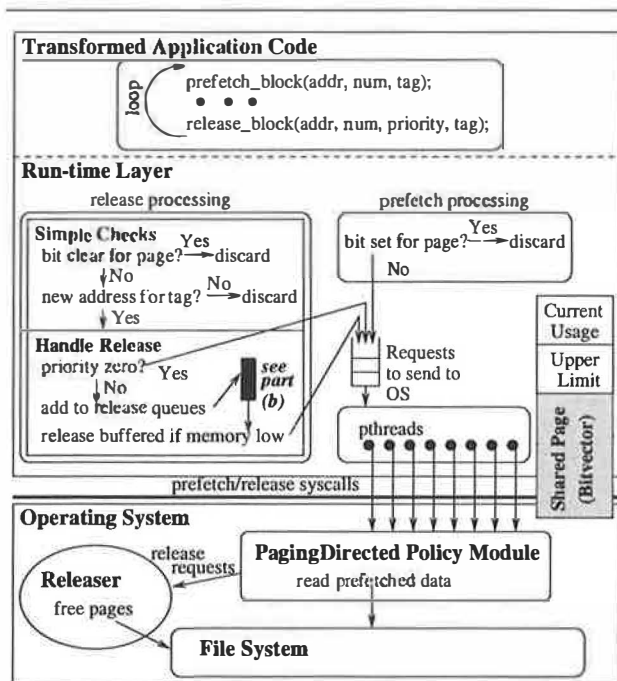
(b) Code with Prefetch and Release

```
for (iter = 0; iter < 10; iter++) {
  for (i = 0; i < 100; i++) {
    prefetch_block(&a[i][0], 56, 1, 0);
    prefetch_block(&b[0], 56, 3, 3);
    for (j1 = 0; j1 < 770048; j1 += 16384) {
      prefetch_release_block(&a[i][245759 + j1],
                            &a[i][j1-16384], 4, 1, 2);
      prefetch_release_block(&b[245759 + j1],
                            &b[j1-16384], 4, 3, 5);
      for (j = j1; j < j1 + 16384; j++)
        c[i] = c[i] + a[i][j]*b[j];
    }
    for (j = 770048; j < 1000000; j++)
      c[i] = c[i] + a[i][j]*b[j];
    release_block(&a[i][770048], 56, 1, 1);
    release_block(&b[770048], 56, 3, 4);
  }
}
```

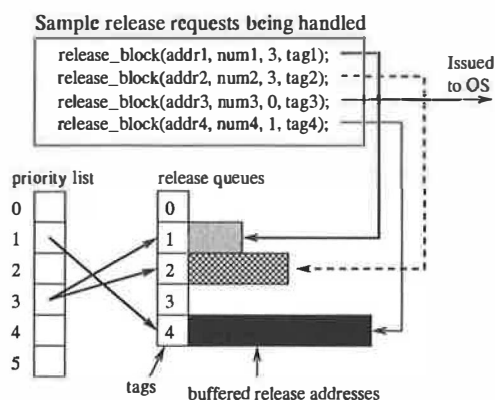
Figure 5. Example of the output of the prefetching compiler. Arguments are: (prefetch address, release address, number of 16KB pages, release priority, request identifier)

3.3 Implementation of the Run-time Layer

Figure 6 illustrates how prefetches and releases are processed by the run-time layer. To achieve the full benefit of prefetching, we need to be able to both fetch data asynchronously (so the application can continue after issuing the prefetch) and take advantage of any available parallelism in the disk subsystem. The run-time layer accomplishes these requirements by creating a number of *threads* [11] that make the actual calls to the Paging-Directed PM and wait for the prefetches to complete. When a prefetch request inserted by the compiler is intercepted by the run-time layer, the bitvector is checked to see if a prefetch is really needed. If so, the request is placed on a work queue and one of the prefetching threads is signaled to handle the request. The prefetching threads simply remove requests from the queue and issue them to



(a) Processing of prefetch and release requests in the run-time layer.



(b) Buffering of release requests using tags and priorities assigned by the compiler.

Figure 6. Handling prefetches and releases at run-time.

the PagingDirected PM. We chose to use a pthreads-based approach since the IRIX kernel does not provide asynchronous I/O to user-level programs. Rather than attempt to add this functionality to IRIX, we chose an approach very similar to the implementation of the asynchronous I/O library in IRIX.

The same set of pthreads are also used to actually issue the release requests to the OS. We have built run-time layers which implement two different policies for handling the release requests inserted by the compiler—one aggressively issues release requests to the OS at the time when they are encountered, while the other buffers releases based on the compiler-inserted priorities and only issues requests when necessary, based on the information provided by the OS. By comparing these two approaches, we can evaluate the usefulness of buffering release requests in the run-time

layer rather than simply relying on the compiler analysis.

In both cases, the run-time layer attempts to reduce overhead by filtering out the obviously bad releases inserted by the compiler. There are two ways in which these bad releases are detected. First, the requests inserted by the compiler are checked against the bitvector to make sure that the pages are in memory. Second, the run-time layer tracks the last address released for each unique release directive placed in the code, using the request identifier (or tag) generated by the compiler. The first release request for any tag is recorded until the next request for that tag is issued. If a release request identifies the same page as the previous request, it is dropped since the page is obviously still in use. If instead, the current release request identifies a different page, then the previously recorded release is actually handled and the current one is recorded. The releases issued by the run-time layer are thus always one or more iterations behind those identified by the compiler. Handling a previously recorded request involves either placing it in a release queue (if buffering is being used), or issuing it to the OS. Programs with loop nests that have unknown bounds often cause the compiler to generate overly-aggressive code, and these simple checks help to reduce the overhead of releasing pages that are still in active use.

Figure 6(b) shows how release requests are buffered. Requests with no reuse (i.e. a priority of 0) are issued to the OS after passing the simple checks. Other requests are stored in release queues indexed by their tags, allowing multiple buffered releases for a particular reference to be coalesced into a single entry in the queue. When the first release for a tag is seen, the priority value is used to index into the priority list where a pointer is set to the release queue for that tag. The priority list can hold pointers to multiple queues having the same priority. When a release request is placed into one of the queues, the current memory usage and memory limit are checked. If the current usage is close to the limit, the priority list is used to issue releases from the lowest-priority queues. Requests are issued from all queues at the same priority level in a round-robin fashion. Currently, the run-time layer attempts to release a total of 100 pages whenever releasing is deemed necessary. (We have not experimented with varying this parameter.)

As we will show in Section 4, even the simple strategy of always issuing the releases improves the performance of the prefetching out-of-core application over prefetching alone, while simultaneously keeping memory free for other applications in most cases. When there is temporal reuse in an application, however, the advantages of prioritizing releases become clear.

4 Experimental Results

To evaluate the concepts presented in this paper, we ran several out-of-core applications with the simulated interactive task described in Section 1.1. We will first describe the platform used to obtain these results, then look at the impact of prefetching, alone and with both aggressive releasing and release buffering, on the execution time of the

Table 1. Experimental platform characteristics.

Processor	
Processor type:	MIPS R10000
Number of Processors:	4
Clock rate:	180 MHz
Physical Memory	
Total size:	128 MBytes
Available to application:	75 MBytes
Page size:	16 KBytes
Disks	
Manufacturer:	Seagate
Model:	Cheetah 4LP
Number of disks used for swap:	10
Maximum external (I/O) transfer rate:	40 Mbytes/sec/disk
Average rotational latency:	2.99 msec
Track-to-track seek, read:	18 msec (typical)
Track-to-track seek, write:	19 msec (typical)
Number of SCSI controllers:	5
Disks per controller:	2

out-of-core program. To explain the basic performance results, we will then take a closer look at the effectiveness of the release operation by examining the activity in the virtual memory subsystem. Finally, we evaluate the usefulness of explicitly releasing memory for improving the response time of the interactive task.

4.1 Hardware Platform

Our experimental results were obtained on a 4-processor SGI Origin 200, running our modified version of the IRIX 6.5 operating system. The system was configured so that approximately 75MB of physical memory was available to user programs, and the system swap space was striped across ten Seagate Cheetah 4LP disks using raw swap partitions. Five SCSI adapters each control two of these ten disks; the SCSI adapters are in turn connected to the PCI buses on the Origin. The basic hardware characteristics of our system are summarized in Table 1.

4.2 Benchmarks

We performed our experiments using out-of-core versions of five applications taken from the NAS Parallel benchmark suite [1] as well as a matrix-vector multiplication kernel (MATVEC). The code for MATVEC was shown earlier in Figure 5(a). We have increased the data sets of the NAS benchmarks to make them larger than the available memory on our system. Other than increasing the data set sizes, we did not modify these applications by hand in any way—all prefetch and release operations were inserted automatically by our compiler pass.

Table 2 summarizes the characteristics of these applications; each exhibits different data access behavior. EMBAR has only one-dimensional loops, while MATVEC has multi-dimensional loops with known bounds. For both, the compiler analysis is essentially perfect and excellent results are obtained for both the benchmarks themselves and the interactive task. BUK and CGM are more difficult cases, as they involve both unknown loop bounds and indirect references, both of which reduce the compiler's ability to analyze the data accesses. Nonetheless, the run-time layer is able to

Table 2. Description of applications.

Name	Description	Input Data Set	Memory Required (and % of Available)	Orig Exec. Time (mins)
BUK	integerbucket sort algorithm	2^{24} 20-bit integers	206MB (275%)	13.5
CGM	sparse linear system solver	40k x 40k sparse matrix, ~15M non-zeros	206 MB (275%)	16.2
EMBAR	monte-carlo simulation	2^{24} random numbers	134 MB (179%)	13.8
FFTPDE	3-D FFT PDE	256x128x128 complex matrix	235 MB (313%)	34.2
MGRID	computes 3-D potential using multigrid solver	256x256x256 matrix	452MB (600%)	23.9
MATVEC	matrix-vector multiply	$10^2 \times 10^6$ matrix, 10^6 vector	404MB (539%)	11.1

adapt the behavior based on dynamic conditions and excellent results are again achieved. MGRID and FFTPDE are the most difficult cases. Both involve multi-dimensional loops with unknown bounds. In MGRID the loop bounds change dynamically on different calls to the same procedures, making it impossible to release memory optimally in all cases, since we only generate a single version of the code. In FFTPDE, the access stride changes within a set of loops, making it seem as though the access is not dependent on the loop induction variable. This causes the compiler to identify some releases as having reuse when in fact none exists. Ultimately, the solution to the problems experienced by MGRID and FFTPDE is to generate more adaptive code, and specialize the loops at run-time according to dynamic conditions. Even without this extra sophistication, MGRID performs better with releases and can significantly reduce (although not eliminate) its negative impact on interactive response time. We believe that any additional improvements to the results shown here will come from improved compiler analysis and code generation, and greater run-time layer involvement, rather than from additional operating system support.

4.3 Performance of the Out-of-Core Applications

The goal of I/O prefetching is to improve the execution time of out-of-core applications by hiding the page fault latency. The goals of explicitly releasing memory are to reduce the number of page faults in out-of-core programs by making better replacement decisions, to reduce the interference caused by the OS selecting victims for replacement, and to alleviate the impact of out-of-core programs on other applications sharing the same system. We begin by examining how well our scheme achieves these goals from the perspective of the out-of-core applications.

In Figure 7, we show the execution times of the out-of-core programs, normalized to the original case. For each benchmark we show four bars: the original, unmodified program (O), the program compiled to use prefetching only (P), the program compiled to use both prefetching and aggressive releasing (R), and the program compiled to use both prefetching and release buffering (B). Each bar is broken down into four components. The top section is the time

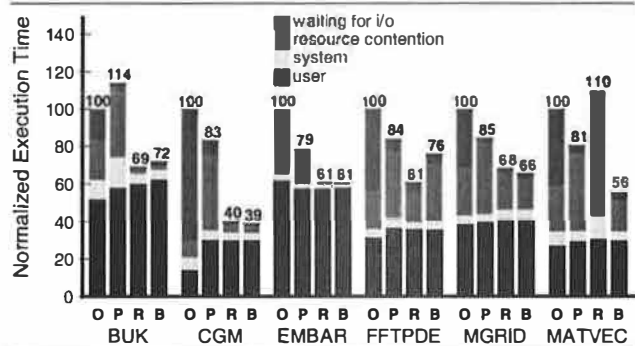


Figure 7. Impact of prefetching and releasing on the execution times of the out-of-core applications. (O = original, P = with prefetching, R = with prefetching and releasing, B = with prefetching and release buffering)

that the program was stalled waiting for I/O. The next component is the time that the process was stalled waiting for unavailable resources, including physical memory, memory system locks, and CPUs. The second-lowest component is the system time, which is primarily spent handling page faults. The bottom section of each bar is the time spent executing user code. Increases in user time over the original case show the overhead of handling prefetch and release requests in the run-time layer. Because we use separate threads to issue the prefetch requests, the prefetch service time does not appear in the execution time of the main application. Since we are using a multiprocessor, many of the prefetches can be serviced in parallel. Although the prefetch threads compete with the main application and the interactive task for CPU time, it is a very small effect since these threads spend most of their time waiting for I/O.

All prefetching versions of the benchmarks achieve similar reductions in the I/O stall time, with over 85% of the I/O stall eliminated in all cases. The time spent executing system code is nearly identical across all versions of the benchmarks, and only modest increases in user time occur in the prefetching versions. The increase in user time is most pronounced for CGM, where a very large number of unnecessary prefetch and release requests need to be filtered out by the run-time layer. These unnecessary requests are the result of the compiler's inability to reason about the amount of data accessed in loops with unknown bounds. For CGM, most of these loops are small and prefetches and releases are not needed. In all cases except for FFTPDE and MATVEC, the results for aggressive releasing and release buffering are very similar, since these applications do not have temporal reuse within a single set of loops, and the compiler analysis is unable to detect reuse across independent sets of loops. When all release requests have zero-priority, both implementations of the run-time layer perform the same actions (issuing the requests to the OS without buffering), although the version which attempts to buffer requests incurs a small amount of additional overhead to check the priorities. In FFTPDE, the compiler incorrectly identifies some references as having temporal reuse, causing the run-time layer to preferentially

retain these pages in memory to the detriment of others. For MATVEC, however, the benefit of buffering and prioritizing releases is dramatic. In this case, without buffering, both the matrix and the vector are released, but the vector is frequently reused shortly thereafter. Large amounts of contention occur between the release daemon attempting to free the pages of the vector and the application attempting to reclaim them. When the run-time layer buffers and prioritizes the releases, only the pages of the matrix need to be released and contention is greatly reduced. In the remainder of this section, we will discuss both releasing versions of the benchmarks together, since their behavior is essentially the same, making specific reference to MATVEC in the cases where buffering makes a difference.

The I/O stall reductions, and the system time and user time components of these experiments all validate the results we obtained in our previous study on compiler-based I/O prefetching [15], demonstrating that these techniques are still applicable with modern hardware and software. Our prior study, however, showed that releasing memory provided no significant benefit to the out-of-core applications over prefetching alone. One key difference here is that the earlier compiler implementation did not insert release operations in many situations. Our results here, in contrast, show that there is a substantial reduction in the execution time of the out-of-core applications when releasing is applied aggressively. The speedups from applying both prefetching and releasing over prefetching alone range from 13% for EMBAR to over 50% for CGM. This added benefit is rather unexpected, both because it did not occur in the previous study, and because the run-time layer implementations are not trying to actively improve the replacement policy (since there is no known reuse)—they simply try to maintain as large a pool of free memory as possible by releasing pages which the application apparently no longer needs. There are essentially three reasons for the improvement due to aggressive releasing: (i) a reduction in the number of soft page faults caused by the paging daemon attempting to identify unused pages; (ii) a reduction in the contention for memory locks needed by both the fault handling code and the paging daemon; and (iii) improvements in the replacement policy created by the compiler analysis alone. We now discuss the impact of each of these effects.

Looking at the components of the bars in Figure 7, we see that the greatest difference between the prefetching-only and the two prefetching-and-releasing cases is in the time stalled for unavailable resources. Without releasing, the paging daemon needs to determine which pages should be reclaimed. To do so, a variant of a clock algorithm is used, in which pages can be reclaimed if they have not been referenced for a number of passes of the clock hand. Since the MIPS TLB does not have reference bits, reference information must be simulated in software using the valid bit instead. As free memory becomes low, pages are periodically marked invalid to see if they are still in use. These invalidations increase the number of soft page faults as the process references, and needs to re-validate, the pages that

Table 3. Pages freed by system or by release, and pages rescued from the free list.

Benchmark	Original				With Prefetch and Release					
	Pages Stolen by System	System Page Reclamation Events	Stolen Pages Rescued	Total Pages Allocated	Pages Stolen by System	System Page Reclamation Events	Stolen Pages Rescued	Pages Freed by release	Released Pages Rescued	Total Pages Allocated
BUK	126,842	2,796	32,532	131,354	5,043	111	4,340	33,916	3,176	158,210
CGM	289,696	6,130	3,472	313,522	1,567	34	109	72,276	266	305,805
EMBAR	126,793	2,987	4	165,838	0	0	0	32,712	4	132,170
FFTPDE	330,490	7,847	9,999	389,504	134,612	3,172	16,574	81,520	2,801	395,478
MGRID	313,595	7,555	806	376,301	72,883	1,735	111	255,114	183,835	360,599
MATVEC	272,541	11,679	7,159	281,297	0	0	0	105,588	261,100	286,294

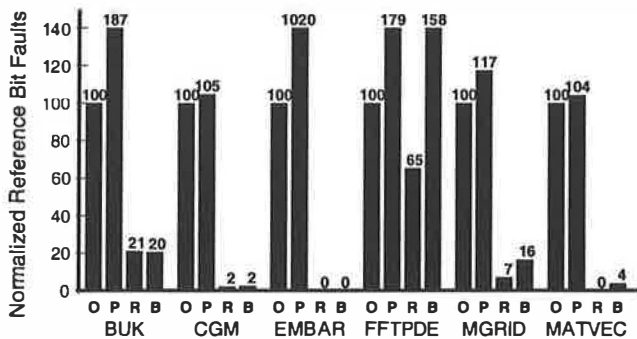


Figure 8. Soft page faults due to page invalidations.

were still in its working set. However, with aggressive releasing, the paging daemon does not need to find pages to reclaim, thus greatly reducing the number of invalidations.

Figure 8 shows the number of page faults caused by these periodic invalidations for each version of our out-of-core benchmarks. Not only are the number of soft page faults greater when prefetching is used without releasing, the time to service each of these faults is also amplified due to increased contention for locks between the paging daemon and the fault handling code. The time to handle hard page faults is also increased by this contention. When the paging daemon needs to invalidate or reclaim pages, it holds locks on the address spaces of the processes from which pages are being stolen. During this time, page faults for these virtual memory regions cannot be serviced. The releasing daemon must hold the same locks while freeing the explicitly released pages; however, it typically operates on smaller blocks of pages, so the locks can be held for much shorter periods of time. Furthermore, the releasing daemon has been specialized for the purpose of freeing pre-identified pages. Thus, it requires fewer locks overall and can do much less processing per page while locks are held. The resulting lock contention caused by the releasing daemon is significantly less than that caused by the paging daemon.

Finally, in some cases the compiler analysis is able to improve upon the replacement policy without extra support from the run-time layer. In BUK, the data set consists of two very large sequentially-accessed arrays and a third equally large randomly-accessed array. The compiler inserts releases for the first two, but does not try to release the third because it cannot reason about any locality that may exist. The result is that demand for new pages is satisfied by the releases of the first two arrays and the pages of

the third array are able to remain mostly in memory. Without releasing, the paging daemon reclaims pages from all three arrays according to their last use, but without regard to their access patterns, causing many more page faults to occur. Although the run-time layer is not able to prioritize releases due to a lack of temporal reuse, the decision by the compiler to not release randomly accessed data effectively accomplishes the desired effect. Having discussed the overall performance impact of our system, we now take a closer look at how effective the compiler and run-time layer are at generating and managing releases.

4.4 Effectiveness of Releases

There are two considerations when evaluating the effectiveness of the release operation. First, the purpose of issuing releases is to maintain a large enough pool of free memory to prevent the default page reclamation behavior. To see how well we achieve this goal, we look at how much work the paging daemon performs, both with and without releases. Second, we should only be releasing pages that are really no longer in use by the application (or will not be used again for a long time) to avoid increasing the page fault rate. To see how useful the releases are, we look at how many released pages are “rescued” from the free list (i.e. returned to the process that was using it). If we are actually releasing pages that are no longer needed, very few pages should be rescued. The page reclamation and allocation activity is summarized in Table 3 for the original out-of-core programs and the versions that both prefetch and release memory without buffering.

From Table 3, we see that releases are usually very effective at reducing the need for the paging daemon to reclaim memory. In the worst case, the number of times that the paging daemon needs to operate is reduced by more than half, and the total number of pages stolen is reduced by more than a factor of three. In the other cases, the activity of the paging daemon is reduced by one to two orders of magnitude, both in terms of frequency and number of pages stolen. Although it is very difficult for the application to release its pages perfectly, it can still provide a great deal of assistance to the OS.

Next we look at how often useful pages are reclaimed too early, either by the paging daemon or due to explicit release requests. There are two possibilities. First, useful pages may still be on the free list when they are referenced again, and can be rescued and returned to the application. Second, useful pages may have been re-allocated to hold other data before being referenced again, and the reused

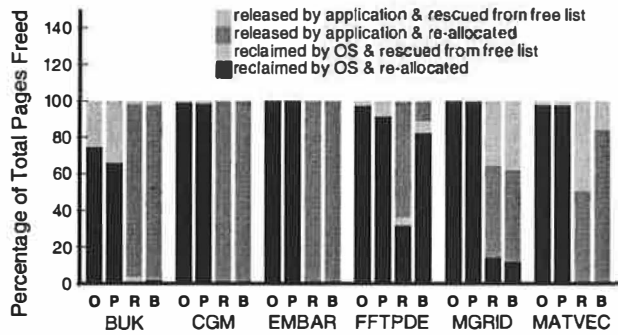
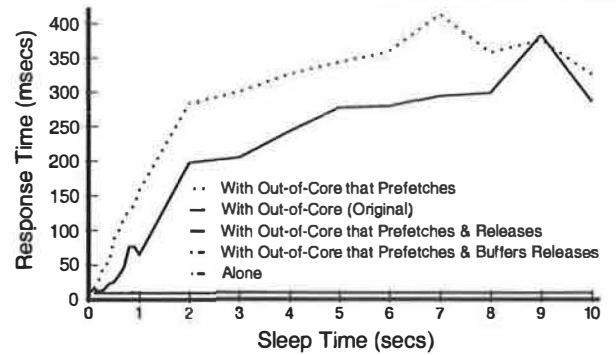


Figure 9. Breakdown of outcomes for freed pages.

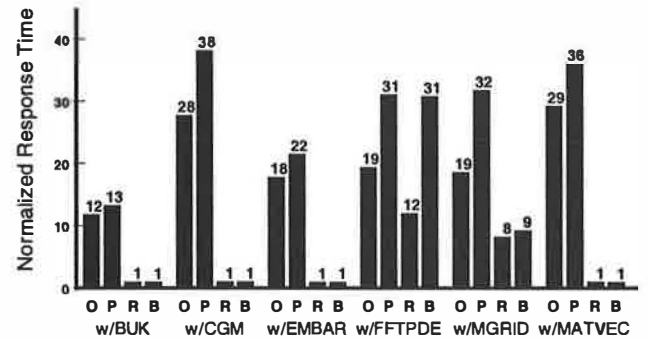
data will need to be brought back into memory from swap.

Figure 9 shows what fraction of all the pages freed are freed by the paging daemon vs. the fraction freed explicitly by release requests. We also show the fraction of each that are rescued from the free list. The interesting cases here are BUK, MGRID and MATVEC. As we see in Figure 9, BUK without any releasing (both the original and prefetching versions) frequently needs to rescue the pages reclaimed by the paging daemon from the free list. The greater demand on memory introduced by prefetching increases the need for the paging daemon to reclaim memory, resulting in useful pages being placed on the free list more often. Consequently, the fraction of reclaimed pages that are rescued also increases. With releasing, however, most of the pages are freed by explicit release requests and very few are rescued from the free list. In this case, releasing helps the application to retain its most-needed pages in memory. For MGRID, we see that even with releasing, over half of the pages freed are reclaimed by the paging daemon, and that more than half of the pages explicitly released are rescued from the free list. This suggests that the compiler is unable to determine which pages to release and when for MGRID. Note also that FFTPDE with release buffering performs very few useful releases due to incorrectly attempting to retain pages with no reuse. For MATVEC without releasing, the OS does a reasonable job of freeing the pages of the matrix and keeping the frequently accessed vector in memory. With aggressive releasing, however, approximately half of the pages released are for the vector and need to be rescued from the free list. When release buffering is used, most of the released pages are for the matrix, and the number of rescued pages is much smaller. Overall, we can see that releasing greatly reduces the need for the paging daemon to reclaim memory, and typically does a good job of releasing pages that are no longer in use.

Detecting pages that were freed too early and re-allocated before they could be rescued is a more difficult task. These pages will increase the total number of page allocations required (over the ideal) as new pages are needed to bring the reused data back into memory. While we cannot compare the total number of page allocations to the ideal number, we can look at the number of allocations



(a) Impact of MATVEC on response time (last 3 lines in key overlap).



(b) Impact of each out-of-core benchmark on response time at 5 second sleep time, normalized to stand-alone response time of 9.5 msec.

Interactive with Benchmark	Hard Page Faults			
	Original	Prefetch Only	Prefetch & Release	Prefetch & Buffer Release
BUK	25	29	0	0
CGM	61	65	0	0
EMBAR	51	62	0	0
FFTPDE	28	55	28	44
MGRID	30	48	11	11
MATVEC	61	63	0	0

(c) Average number of page faults requiring I/O for the interactive task with each out-of-core benchmark.

Figure 10. Impact of releasing on interactive response time.

in the original case versus the prefetching-and-releasing cases. From Table 3, we see that the total number of page allocations increases by a small amount with prefetching and releasing in half of the cases, and decreases by a small amount in the other half. This suggests that releasing is typically doing no worse at freeing needed pages than the paging daemon, but results in much less contention.

We now look at how useful releases are for improving the performance of the interactive task.

4.5 Impact on Interactive Response Time

Figure 10 gives an overview of the performance improvements obtained for the “interactive” task. In Figure 10(a), we show the average response time for the interactive task when executed concurrently with MATVEC across a range of sleep times. As discussed in Section 1.1, the response times become greatly inflated when the out-of-

core program executes normally, and are made even worse when prefetching alone is used. When releasing is added to prefetching, however, the response times of the interactive task almost perfectly matches the times obtained when it is run alone on the machine, regardless of the amount of sleep time. Although blindly following the release directives inserted by the compiler has a severe effect on MATVEC's own performance, this strategy does leave most of memory free for the interactive task. However, when release buffering is used to improve the performance of MATVEC, there is still nearly no impact on the interactive task. The run-time layer is able to both buffer releases for the benefit of the out-of-core task and keep enough memory free for the interactive one. The negative impact of the out-of-core program on the response time of the interactive task in this case has been almost completely eliminated. For the other out-of-core applications, we chose an intermediate sleep time of five seconds for the interactive task and recorded the average response times. The results for each of the four versions of the out-of-core programs are shown in Figure 10(b). The response times in this graph have been normalized to the time for the interactive task executing alone on the machine. As we see in Figure 10(b), releasing is usually successful at eliminating or substantially reducing the degradation in interactive response time. FFTPDE with release buffering is the exception as this benchmark fails to release enough memory.

Figure 10(c) shows the average number of hard page faults (i.e. those that require I/O) experienced by the interactive task during a single sweep through its data set, when it is executed concurrently with each version of our out-of-core benchmarks. From this table, we see that the number of page faults increases when the out-of-core program uses prefetching alone, rising to the maximum level of 65 pages. At this point, the entire data set of the interactive task must be paged in from the swap space. When the out-of-core program also releases pages, the number of hard page faults is significantly reduced. This result verifies that the primary reason for the increased interactive response time is not being able to keep pages in memory.

5 Related Work

Many researchers have suggested that better performance can be obtained if sophisticated applications are given control over their own memory management decisions. Most previous work in this area has focused on how the OS can provide this functionality to the applications. For instance, the Mach operating system supports external pagers to allow applications to control the backing storage of their memory objects [18]. Extensions to the external pager interface have been used to implement user-level page replacement policies [14] and to support discardable pages (i.e. dirty pages that do not have to be written to backing store) [20]. More aggressive application control of physical memory was implemented in the V++ kernel by Harty and Cheriton [10]. In their scheme, the application was given complete control over a cache of physical

pages, enabling the implementation of application-specific memory management policies. Giving applications more control over physical resources (not just memory) is also a part of the motivation behind extensible operating systems such as Exokernel [12], SPIN [2], and Vino [19]. Providing support for application-specific control is only half of the picture, however. If the mechanisms provided require programmers to re-write their applications manually, the full power of the scheme is unlikely to be realized in the real world. In contrast, our approach provides not only the mechanisms for application-controlled memory management, but also a means to leverage these mechanisms automatically through the use of the compiler.

Other related work has shown the importance of considering both prefetching and replacement decisions in tandem, in the context of I/O prefetching for file system references. Cao *et al.* [3] present several properties that optimal prefetching and caching strategies must have; however the complete reference stream is required to satisfy these properties. The TIP system for I/O prefetching by Patterson *et al.* [16] uses a cost-benefit model to estimate which file blocks should be replaced from the buffer cache, based on access-pattern hints disclosed by the application. While the goal of using application-specific knowledge to improve overall system performance is the same as in our system, we focus on virtual memory references rather than file reads and writes. In the original TIP implementation, applications had to be manually modified to generate the necessary access hints. Recently, another approach for automatically modifying applications to provide hints about their future accesses has been presented by Chang and Gibson [4]. Applications are modified automatically (using a binary modification tool on the program executable) to speculatively execute the code and generate access pattern hints to be passed to the TIP system. Because it is much more costly to track all virtual memory references (versus explicit file requests only) the techniques used by the TIP system for deciding what to eject from the file cache are not especially applicable for virtual memory management.

6 Conclusions

We have implemented and evaluated a complete and fully-automatic system which exploits compiler-inserted *release* operations to intelligently manage the physical memory resources of out-of-core applications. These specialized applications can reduce their impact on the performance of other applications while still exploiting aggressive prefetching to hide their I/O latency. Our results confirm that compiler-inserted I/O prefetching works well on commercial operating systems and state-of-the-art machines (even though faster processors make it much more challenging to hide the I/O latency), hiding roughly 85-100% of the I/O stall time in our out-of-core benchmarks and achieving good overall speedups.

The significant benefit to the out-of-core benchmarks due to aggressively releasing memory was mostly unexpected. In BUK we expected to see a benefit from improv-

ing on the replacement policy, but for the other applications (excepting MATVEC, which is hurt by aggressive releasing), the improvement comes from reducing the interference between the operating system and the application. We found the extent of this interference between the paging daemon and the page fault handling to be especially surprising. Not only does the paging daemon greatly increase the number of soft page faults as it attempts to simulate reference bits in software, but the time to handle these page faults is also inflated by increased lock contention. Because the overhead of determining which pages to replace is so large, explicit replacement hints can improve performance, even if they are not making better replacement decisions than the default policy. It would be interesting to see if these benefits still occur on a system with hardware reference bits (although such a study was beyond the scope of this paper since IRIX only runs on MIPS processors).

Overall, our compiler-based approach for combining both prefetching and releasing to allow out-of-core applications to explicitly manage their virtual memory is a situation in which everyone wins. Both the memory-intensive programs and the less demanding interactive ones sharing the system obtain performance benefits. Only the out-of-core programs need to be modified, and the changes are performed automatically by the compiler without burdening the application programmer. Furthermore, the default policies of the operating system do not need to be changed, and no overhead is introduced in the common case for managing ordinary applications.

7 Acknowledgements

We thank Andrew Myers (our shepherd) for helping us improve the presentation of this paper. This research is supported by a grant from NASA. Todd C. Mowry is partially supported by an Alfred P. Sloan Research Fellowship.

References

- [1] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS Parallel Benchmarks. Technical Report RNR-91-002, NASA Ames Research Center, Aug. 1991.
- [2] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Ficzynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proc. of the 15th Symp. on Operating System Principles*, Dec. 1995.
- [3] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A Study of Integrated Prefetching and Caching Strategies. In *Proc. of the ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 188–197, 1995.
- [4] F. Chang and G. Gibson. Automatic I/O Hint Generation Through Speculative Execution. In *Proc. of the 3rd OSDI*, Feb. 1999.
- [5] M. Cox and D. Ellsworth. Application-Controlled Demand Paging for Out-of-Core Visualization. In *Proc. of Visualization '97*, Oct. 1997.
- [6] P. E. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed. Input/Output Characteristics of Scalable Parallel Applications. In *Proc. of Supercomputing '95*, Dec. 1995.
- [7] J. M. del Rosario and A. Choudhary. High Performance I/O for Massively Parallel Computers: Problems and Prospects. *IEEE Computer*, 27(3):59–68, Mar. 1994.
- [8] A. K. Demke. Automatic I/O Prefetching for Out-of-Core Applications. Master's thesis, University of Toronto, Department of Computer Science, Jan. 1997.
- [9] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, 29(12):84–89, Dec. 1996.
- [10] K. Harty and D. Cheriton. Application-Controlled Physical Memory Using External Page-Cache Management. In *Proc. of the 5th ASPLOS*, pages 187–199, Oct. 1992.
- [11] IEEE. Threads Extension for Portable Operating Systems (Draft 7), Feb. 1992.
- [12] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceo, R. Hunt, D. Mazires, T. Pinckney, R. Grimm, J. Jannotti, and K. MacKenzie. Application Performance and Flexibility on Exokernel Systems. In *Proc. of the 16th Symp. on Operating System Principles*, Oct. 1997.
- [13] C.-K. Luk and T. C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *Proc. of the 7th ASPLOS*, pages 222–233, Oct. 1996.
- [14] D. McNamee and K. Armstrong. Extending the Mach External Pager Interface to Accommodate User-Level Page Replacement Policies. In *Proc. of the USENIX Assoc. Mach Workshop*, pages 17–29, 1990.
- [15] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications. In *Proc. of the 2nd OSDI*, pages 3–17, Oct. 1996.
- [16] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proc. of the 15th Symp. on Operating System Principles*, pages 79–95, Dec. 1995.
- [17] J. T. Poole. Preliminary Survey of I/O Intensive Applications. Technical Report CCSF-38, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994.
- [18] R. Rashid, A. Tevanian, Jr., M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proc. of the 2nd ASPLOS*, Oct. 1987.
- [19] C. Small and M. Seltzer. A Comparison of OS Extension Technologies. In *Proc. of the 1996 USENIX Technical Conference*, Jan. 1996.
- [20] I. Subramanian. Managing Discardable Pages with an External Pager. In *Proc. of the USENIX Mach Symposium*, Nov. 1991.
- [21] R. C. Unrau, O. Krieger, B. Gamsa, and M. Stumm. Hierarchical Clustering: A Structure for Scalable Multiprocessor Operating System Design. *Journal of Supercomputing*, 9(1/2):105–134, 1995.
- [22] Z. G. Vranesic, M. Stumm, R. White, and D. Lewis. The Hector Multiprocessor. *IEEE Computer*, 24(1), Jan. 1991.

Surplus Fair Scheduling: A Proportional-Share CPU Scheduling Algorithm for Symmetric Multiprocessors *

Abhishek Chandra, Micah Adler, Pawan Goyal[†] and Prashant Shenoy

Department of Computer Science,
University of Massachusetts Amherst
{abhishek,micah,shenoy}@cs.umass.edu

[†]Ensim Corporation
Sunnyvale, CA
goyal@ensim.com

Abstract

In this paper, we present surplus fair scheduling (SFS), a proportional-share CPU scheduler designed for symmetric multiprocessors. We first show that the infeasibility of certain weight assignments in multiprocessor environments results in unfairness or starvation in many existing proportional-share schedulers. We present a novel weight readjustment algorithm to translate infeasible weight assignments to a set of feasible weights. We show that weight readjustment enables existing proportional-share schedulers to significantly reduce, but not eliminate, the unfairness in their allocations. We then present surplus fair scheduling, a proportional-share scheduler that is designed explicitly for multiprocessor environments. We implement our scheduler in the Linux kernel and demonstrate its efficacy through an experimental evaluation. Our results show that SFS can achieve proportionate allocation, application isolation and good interactive performance, albeit at a slight increase in scheduling overhead. We conclude from our results that a proportional-share scheduler such as SFS is not only practical but also desirable for server operating systems.

1 Introduction

1.1 Motivation

The growing popularity of multimedia and web applications has spurred research in the design of large multiprocessor servers that can run a variety of demanding applications. To illustrate, many commercial web sites today employ multiprocessor servers to run a mix of HTTP applications (to service web requests), database applications (to store product and customer information), and streaming media applications (to deliver audio and video

content). Moreover, Internet service providers that host third party web sites typically do so by mapping multiple web domains onto a single physical server, with each domain running a mix of these applications. These example scenarios illustrate the need for designing resource management mechanisms that multiplex server resources among diverse applications in a predictable manner.

Resource management mechanisms employed by a server operating system should have several desirable properties. First, these mechanisms should allow users to specify the fraction of the resource that should be allocated to each application. In the web hosting example, for instance, it should be possible to allocate a certain fraction of the processor and network bandwidth to each web domain [2]. The operating system should then allocate resources to applications based on these user-specified shares. It has been argued that such allocation should be both fine-grained and fair [3, 9, 15, 17, 20]. Another desirable property is application isolation—the resource management mechanisms employed by an operating system should effectively isolate applications from one another so that misbehaving or overloaded applications do not prevent other applications from receiving their specified shares. Finally, these mechanisms should be computationally efficient so as to minimize scheduling overheads. Thus, efficient, predictable and fair allocation of resources is key to designing server operating systems. The design of a CPU scheduling algorithm for symmetric multiprocessor servers that meets these objectives is the subject matter of this paper.

1.2 Relation to Previous Work

In the recent past, a number of resource management mechanisms have been developed for predictable allocation of processor bandwidth [2, 7, 11, 12, 14, 16, 18, 24, 28]. Many of these CPU scheduling mechanisms as well as their counterparts in the network packet scheduling domain [4, 5, 19, 23] associate an intrinsic rate with

*This research was supported in part by a NSF Career award CCR-9984030, NSF grants ANI 9977635, CDA-9502639, Intel, Sprint, and the University of Massachusetts.

each application and allocate resource bandwidth in proportion to this rate. For instance, many recently proposed algorithms such as start-time fair queuing (SFQ) [9], borrowed virtual time (BVT) [7], and SMART [16] are based on the concept of *generalized processor sharing (GPS)*. GPS is an idealized algorithm that assigns a weight to each application and allocates bandwidth fairly to applications in proportion to their weights. GPS assumes that threads can be scheduled using infinitesimally small quanta to achieve weighted fairness. Practical instantiations, such as SFQ, emulate GPS using finite duration quanta. While GPS-based algorithms can provide strong fairness guarantees in uniprocessor environments, they can result in unbounded unfairness or starvation when employed in multiprocessor environments as illustrated by the following example.

Example 1 Consider a server that employs the start-time fair queuing (SFQ) algorithm [9] to schedule threads. SFQ is a GPS-based fair scheduling algorithm that assigns a weight w_i to each thread and allocates bandwidth in proportion to these weights. To do so, SFQ maintains a counter S_i for each application that is incremented by $\frac{q}{w_i}$ every time the thread is scheduled (q is the quantum duration). At each scheduling instance, SFQ schedules the thread with the minimum S_i on a processor. Assume that the server has two processors and runs two compute-bound threads that are assigned weights $w_1 = 1$ and $w_2 = 10$, respectively. Let the quantum duration be $q = 1\text{ms}$. Since both threads are compute-bound and SFQ is work-conserving,¹ each thread gets to continuously run on a processor. After 1000 quanta, we have $S_1 = \frac{1000}{1} = 1000$ and $S_2 = \frac{1000}{10} = 100$. Assume that a third cpu-bound thread arrives at this instant with a weight $w_3 = 1$. The counter for this thread is initialized to $S_3 = 100$ (newly arriving threads are assigned the minimum value of S_i over all runnable threads). From this point on, threads 2 and 3 get continuously scheduled until S_2 and S_3 “catch up” with S_1 . Thus, although thread 1 has the same weight as thread 3, it starves for 900 quanta leading to unfairness in the scheduling algorithm. Figure 1 depicts this scenario.

Many recently proposed GPS-based algorithms such as stride scheduling [28], weighted fair queuing (WFQ) [18] and borrowed virtual time (BVT) [7] also suffer from this drawback when employed for multiprocessors (like SFQ, stride scheduling and WFQ are instantiations of GPS, while BVT is a derivative of SFQ with an additional latency parameter; BVT reduces to SFQ when the latency parameter is set to zero). The primary reason for this inadequacy is that while *any arbitrary weight assignment is feasible for uniprocessors, only certain*

weight assignments are feasible for multiprocessors. In particular, those weight assignments in which the bandwidth assigned to a single thread exceeds the capacity of a processor are infeasible (since an individual thread cannot consume more than the bandwidth of a single processor). In the above example, the second thread was assigned $\frac{10}{11}$ of the total bandwidth on a dual-processor server, whereas it can consume no more than half the total bandwidth. Since GPS-based work-conserving algorithms do not distinguish between feasible and infeasible weight assignments, unfairness can result when a weight assignment is infeasible. In fact, even when the initial weights are carefully chosen to be feasible, blocking events can cause the weights of the remaining threads to become infeasible. For instance, a feasible weight assignment of 1:1:2 on a dual-processor server becomes infeasible when one of the threads with weight 1 blocks. Even when all weights are feasible, an orthogonal problem occurs when frequent arrivals and departures prevent a GPS-based scheduler such as SFQ from achieving proportionate allocation. Consider the following example:

Example 2 Consider a dual-processor server that runs a thread with weight 10,000 and 10,000 threads with weight 1. Assume that short-lived threads with weight 100 arrive every 100 quanta and run for 100 quanta each. Note that the weight assignment is always feasible. If SFQ is used to schedule these threads, then it will assign the current minimum value of S_i in the system to each newly arriving thread. Hence, each short-lived thread is initialized with the lowest value of S_i and gets to run continuously on a processor until it departs. The thread with weight 10,000 runs on the other processor; all threads with weight 1 run infrequently. Thus, each short-lived thread with weight 100 gets as much processor bandwidth as the thread with weight 10,000 (instead of $\frac{1}{100}$ of the bandwidth). Note that this problem does not occur in uniprocessor environments.

The inability to distinguish between feasible and infeasible weight assignments as well as to achieve proportionate allocation in the presence of frequent arrivals and departures are fundamental limitations of a proportional-share scheduler such as SFQ. Whereas randomized schedulers such as lottery scheduling [27] do not suffer from starvation problems due to infeasible weights, such weight assignments can, nevertheless, cause small inaccuracies in proportionate allocation for such schedulers. Several techniques can be employed to address the problem of infeasible bandwidth assignments. In the simplest case, processor bandwidth could be assigned to applications in absolute terms instead of using a relative mechanism such as weights (e.g., assign 20% of the bandwidth on a processor to a thread). A

¹A scheduling algorithm is said to be work-conserving if it never lets a processor idle so long as there are runnable threads in the system.

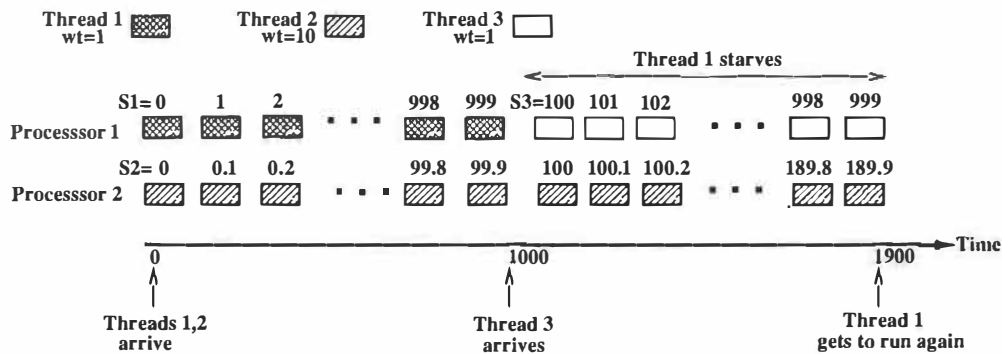


Figure 1: The Infeasible Weights Problem: an infeasible weight assignment can lead to unfairness in allocated shares in multiprocessor environments.

potential limitation of such absolute allocations is that bandwidth unused by an application is wasted, resulting in poor resource utilization. To overcome this drawback, most modern schedulers that employ this method reallocate unused processor bandwidth to needy applications in a fair-share manner [10, 14]. In fact, it has been shown that relative allocations using weights and absolute allocations with fine-grained reassignment of unused bandwidth are duals of each other [22]. A more promising approach is to employ a GPS-based scheduler for each processor and partition the set of threads among processors such that each processor is load balanced. Such an approach has two advantages: (i) it can provide strong fairness guarantees on a per-processor basis, and (ii) binding a thread to a processor allows the scheduler to exploit processor cache locality. A limitation of the approach is that periodic repartitioning of threads may be necessary since blocked/terminated threads can cause imbalances across processors, which can be expensive. Nevertheless, such an approach has been successfully employed to isolate applications from one another [1, 8, 26].

In summary, GPS-based fair scheduling algorithms or simple modifications thereof are unsuitable for fair allocation of resources in multiprocessor environments. To overcome this limitation, we propose a CPU scheduling algorithm for multiprocessors that: (i) explicitly distinguishes between feasible and infeasible weight assignments and (ii) achieves proportionate allocation of processor bandwidth to applications.

1.3 Research Contributions of this Paper

In this paper, we present *surplus fair scheduling (SFS)*, a predictable CPU scheduling algorithm for symmetric multiprocessors. The design of this algorithm has led to several key contributions.

First, we have developed a weight readjustment algorithm to explicitly deal with the problem of infeasible

weight assignments; our algorithm translates a set of infeasible weights to the “closest” feasible weight assignment, thereby enabling all scheduling decisions to be based on feasible weights. Our weight readjustment algorithm is a novel approach for dealing with infeasible weights and one that can be combined with most existing GPS-based scheduling algorithms; doing so enables these algorithms to vastly reduce the unfairness in their allocations for multiprocessor environments. However, even with the readjustment algorithm, many GPS-based algorithms show unfairness in their allocations, especially in the presence of frequent arrival and departures of threads. To overcome this drawback, we develop the surplus fair scheduling algorithm for proportionate allocation of bandwidth in multiprocessor environments. A key feature of our algorithm is that it does not require the quantum length to be known a priori, and hence can handle quanta of variable length.

We have implemented the surplus fair scheduling algorithm in the Linux kernel and have made the source code available to the research community. We have experimentally demonstrated the benefits of our algorithm over a GPS-based scheduler such as SFQ using sample applications and benchmarks. Our experimental results show that surplus fair scheduling can achieve proportionate allocation, application isolation and good interactive performance for typical application mixes, albeit at the expense of a slight increase in the scheduling overhead. Together these results demonstrate that a proportional-share CPU scheduling algorithm such as surplus fair scheduling is not only practical but also desirable for server operating systems.

The rest of this paper is structured as follows. Section 2 presents the surplus fair scheduling algorithm. Section 3 discusses the implementation of our scheduling algorithm in Linux. Section 4 presents the results of our experimental evaluation. Section 5 presents some limitations of our approach and directions for future work.

Finally, Section 6 presents some concluding remarks.

2 Proportional-Share CPU Scheduling for Multiprocessor Environments

Consider a multiprocessor server with p processors that runs t threads. Let us assume that a user can assign any arbitrary weight to a thread. In such a scenario, a thread with weight w_i should be allocated $(w_i / \sum_j w_j)$ fraction of the total processor bandwidth. Since weights can be arbitrary, it is possible that a thread may request more bandwidth than it can consume (this occurs when the requested fraction $\frac{w_i}{\sum_j w_j} > \frac{1}{p}$). The CPU scheduler must somehow reconcile the presence of such infeasible weights. To do so, we present an optimal weight readjustment algorithm that can efficiently translate a set of infeasible weights to the “closest” feasible weight assignment. By running this algorithm every time the weight assignment becomes infeasible, the CPU scheduler can ensure that all scheduling decisions are always based on a set of feasible weights. Given such a weight readjustment algorithm, we then present *generalized multiprocessor sharing (GMS)*—an idealized algorithm for fair, proportionate bandwidth allocation that is an analogue of GPS in the multiprocessor domain. We use the insights provided by GMS to design the *surplus fair scheduling (SFS)* algorithm. SFS is a practical instantiation of GMS that has lower implementation overheads.

In what follows, we first present our weight readjustment algorithm in Section 2.1. We present generalized multiprocessor sharing in Section 2.2 and then present the surplus fair scheduling algorithm in Section 2.3.

2.1 Efficient, Optimal Weight Readjustment

As illustrated in Section 1.2, weight assignments in which a thread requests a bandwidth share that exceeds the capacity of a processor are infeasible. Moreover, a feasible weight assignment may become infeasible or vice versa whenever a thread blocks or becomes runnable. To address these problems, we have developed a weight readjustment algorithm that is invoked every time a thread blocks or becomes runnable. The algorithm examines the set of runnable threads to determine if the weight assignment is feasible. A weight assigned to a thread is said to be feasible if

$$\frac{w_i}{\sum_j w_j} \leq \frac{1}{p} \quad (1)$$

We refer to Equation 1 as the *feasibility constraint*. If a thread violates the feasibility constraint (i.e., requests a fraction that exceeds $1/p$), then it is assigned a new

weight so that its requested share reduces to $1/p$ (which is the maximum share an individual thread can consume). Doing so for each thread with infeasible weight ensures that the new weight assignment is feasible.

Conceptually, the weight readjustment algorithm proceeds by examining each thread in descending order of weights to see if it violates the feasibility constraint. Each thread that does so is assigned the bandwidth of an entire processor, which is the maximum bandwidth a thread can consume. The problem then reduces to checking the feasibility of scheduling the *remaining* threads on the *remaining* processors. In practice, the readjustment algorithm is implemented using recursion—the algorithm recursively examines each thread to see if it violates the constraint; the recursion terminates when a thread that satisfies the constraint is found. The algorithm then assigns a new weight to each thread that violates the constraint such that its requested fraction equals $1/p$. This is achieved by computing the average weight of all feasible threads over the remaining processors and assigning it to the current thread (i.e., $w_i = \frac{\sum_{j=i+1}^p w_j}{p-i}$). Figure 2 illustrates the complete weight readjustment algorithm.

Our weight readjustment algorithm has the following salient features:

- The algorithm is *optimal* in the sense that it changes the weights of the minimum number of threads and the new weights are the “closest” weights that reflect the original assignment. This is because threads with infeasible weights are assigned the nearest feasible weight, and weights of threads that satisfy the feasibility constraint *never* change (and hence, they continue to receive bandwidth in their requested proportions).
- The algorithm has an *efficient* implementation. To see why, observe that in a p -processor system, no more than $(p - 1)$ threads can have infeasible weights (since the sum of the requested fractions is 1, no more than $(p - 1)$ threads can request a fraction that exceeds $\frac{1}{p}$). Thus, the number of threads with infeasible weights depends solely on the number of processors and is independent of the total number of threads in the system. By maintaining a list of threads sorted in descending order of their weights, the algorithm needs to examine no more than the first $(p - 1)$ threads with the largest weights. In fact, the algorithm can stop scanning the sorted list at the first point where the feasibility constraint is satisfied (subsequent threads have even smaller weights and hence, request smaller and feasible fractions). Since the number of processors is typically much smaller than the number of threads


```

readjust(array  $w[1..t]$ , int  $i$ , int  $p$ )
begin
  if(  $\frac{w[i]}{\sum_{j=i}^t w[j]} > \frac{1}{p}$  )
    begin
      readjust( $w[1..t]$ ,  $i + 1$ ,  $p - 1$ )
       $sum = \sum_{j=i+1}^t w[j]$ 
       $w[i] = \frac{sum}{p-1}$ 
    end
  end.

```

Figure 2: The weight readjustment algorithm: The algorithm is invoked with an array of weights sorted in decreasing order. Initially, $i = 1$; p denotes the number of processors, and t denotes the number of runnable threads. If a thread violates the feasibility constraint, then the algorithm is recursively invoked for the remaining threads and the remaining processors. Each infeasible weight is then adjusted by setting its requested processor share to $1/p$.

($p \ll t$), the overhead imposed by the readjustment algorithm is small.

- Our weight readjustment algorithm can be employed with most existing GPS-based scheduling algorithms to deal with the problem of infeasible weights. We experimentally demonstrate in Section 4.2 that doing so enables these schedulers to significantly reduce (but not eliminate) the unfairness in their allocations for multiprocessor environments.

The weight readjustment algorithm can also be employed in conjunction with a randomized proportional-share scheduler such as lottery scheduling [27]. Although such a scheduler does not suffer from starvation problems due to infeasible weights, a set of feasible weights can help such a randomized scheduler in making more accurate scheduling decisions.

Given our weight readjustment algorithm, we now present an idealized algorithm for proportional-share scheduling in multiprocessor environments.

2.2 Generalized Multiprocessor Sharing

Consider a server with p processors each with capacity C that runs t threads. Let the threads be assigned weights $w_1, w_2, w_3, \dots, w_t$. Let ϕ_i denote the instantaneous weight of a thread as computed by the readjustment algorithm. At any instant, depending on whether the thread satisfies or violates the feasibility constraint, ϕ_i is either the original weight w_i or the readjusted weight. From the definition of ϕ_i , it follows that $\sum_j \phi_j \leq \frac{1}{p}$

at all times (our weight readjustment algorithm ensures this property). Assume that threads can be scheduled for infinitesimally small quanta and let $A_i(t_1, t_2)$ denote the CPU service received by thread i in the interval $[t_1, t_2]$. Then the *generalized multiprocessor sharing (GMS)* algorithm has the following property: for any interval $[t_1, t_2]$, the amount of CPU service received by thread i satisfies

$$\frac{A_i(t_1, t_2)}{A_j(t_1, t_2)} \geq \frac{\phi_i}{\phi_j} \quad (2)$$

provided that (i) thread i is continuously runnable in the entire interval, and (ii) both ϕ_i and ϕ_j remain fixed in that interval. Note that the instantaneous weight ϕ remains fixed in an interval if the thread either satisfies the feasibility constraint in the entire interval, or continuously violates the constraint in the entire interval. It is easy to show that Equation 2 implies proportionate allocation of processor bandwidth.²

Intuitively, GMS is similar to a weighted round-robin algorithm in which threads are scheduled in round-robin order (p at a time); each thread is assigned an infinitesimally small CPU quantum and the number of quanta assigned to a thread is proportional to its weight. In practice, however, threads must be scheduled using finite duration quanta so as to amortize context switch overheads. Consequently, in what follows, we present a CPU scheduling algorithm that employs finite duration quanta and is a practical approximation of GMS.

2.3 Surplus Fair Scheduling

Consider a GMS-based CPU scheduling algorithm that schedules threads in terms of finite duration quanta. To clearly understand how such an algorithm works, we first present the intuition behind the algorithm and then provide precise details. Let us assume that thread i is assigned a weight w_i and that the weight readjustment algorithm is employed to ensure that weights are feasible at all times. Let ϕ_i denote the instantaneous weight of thread i . Let $A_i(t_1, t_2)$ denote the amount of CPU service received by thread i in the duration $[t_1, t_2]$, and let $A_i^{GMS}(t_1, t_2)$ denote the amount of service that the thread would have received if it were scheduled using GMS. Then, the quantity

$$\alpha_i = A_i(t_1, t_2) - A_i^{GMS}(t_1, t_2) \quad (3)$$

²This can be observed by summing Equation 2 over all runnable threads j , which yields $A_i(t_1, t_2) \cdot \sum_j \phi_j \geq \phi_i \cdot \sum_j A_j(t_1, t_2)$. Since $\sum_j A_j(t_1, t_2)$ is the total processor bandwidth allocated to all threads in the interval, we can substitute it by the quantity $p \cdot C \cdot (t_2 - t_1)$. Hence, we get $A_i(t_1, t_2) \geq \frac{\phi_i}{\sum_j \phi_j} \cdot p \cdot C \cdot (t_2 - t_1)$. Thus each

thread receives processor bandwidth in proportion to its instantaneous weight ϕ_i .

represents the extra service (i.e., surplus) received by thread i when compared to GMS. To closely emulate GMS, a scheduling algorithm should schedule threads such that the surplus α_i for each thread is as close to zero as possible. Given a p -processor system, a simple approach for doing so is to actually compute α_i for each thread and schedule the p threads with the least surplus values. If the net surplus is negative, doing so allows a thread to catch up with its allocation in GMS. Even when the net surplus of a thread is positive, picking threads with the least positive surplus values enables the algorithm to ensure that the overall deviation from GMS is as small as possible (picking a thread with a larger α_i would cause a larger deviation from GMS).

A scheduling algorithm that actually uses Equation 3 to compute surplus values is impractical since it requires the scheduler to compute A_i^{GMS} (which in turn requires a simulation of GMS). Consequently, we derive an approximation of Equation 3 that enables efficient computation of the surplus values for each thread. Let S_1, S_2, \dots, S_i denote the weighted CPU service received by each thread so far. If thread i runs in a quantum, then S_i is incremented as $S_i = S_i + \frac{q}{\phi_i}$, where q denotes the duration for which the thread ran in that quantum. Since S_i is the weighted CPU service received by thread i , $\phi_i \cdot S_i$ represents the total service received by thread i so far. Let v denote the minimum value of S_i over all runnable threads. Intuitively, v represents the processor allocation of the thread that has received the minimum service so far. Then the surplus service received by thread i is defined to be

$$\alpha_i = \phi_i \cdot (S_i - v) \quad (4)$$

The first term $\phi_i \cdot S_i$ approximates $A_i(0, t)$, which is the service received by thread i so far. The second term $\phi_i \cdot v$ approximates the quantity A_i^{GMS} in Equation 3. Thus, α_i measures the surplus service received by thread i when compared to the thread that has received the least service so far (i.e., v). It follows from this definition of α_i that $\alpha_i \geq 0$ for all runnable threads. Scheduling a thread with the smallest value of α_i ensures that the scheduler approximates GMS and each thread receives processor bandwidth in proportion to its weight. Since a thread is chosen based on its surplus value, we refer to the algorithm as *surplus fair scheduling (SFS)*.

Having provided the intuition for our algorithm, the precise SFS algorithm is as follows:

- Each thread in the system is associated with a weight w_i , a start tag S_i and a finish tag F_i . Let ϕ_i denote the instantaneous weight of a thread as computed by the readjustment algorithm. When a new thread arrives, its start tag is initialized as $S_i = v$, where v is the virtual time of the system (defined

below). When a thread runs on a processor, its finish tag at the end of the quantum is updated as

$$F_i = S_i + \frac{q}{\phi_i} \quad (5)$$

where q is the duration for which the thread ran in that quantum and ϕ_i is its instantaneous weight at the end of the quantum. Observe that q can vary depending on whether the thread utilizes its entire allocated quantum or relinquishes the processor before the quantum ends due to a blocking event. The start tag of a runnable thread is computed as

$$S_i = \begin{cases} \max(F_i, v) & \text{if the thread just woke up} \\ F_i & \text{if the thread is continuously runnable} \end{cases} \quad (6)$$

- Initially, the virtual time of the system is zero. At any instant, the virtual time is defined to be the minimum of the start tags over all runnable threads. The virtual time remains unchanged if all processors are idle and is set to the finish tag of the thread that ran last.
- At each scheduling instance, SFS computes the surplus values of all runnable threads as $\alpha_i = \phi_i \cdot (S_i - v)$ and schedules the thread with the least α_i ; ties are broken arbitrarily.

Our surplus fair scheduling algorithm has the following salient features. First, like most GPS-based algorithms, SFS is work-conserving in nature—the algorithm ensures that a processor will not idle so long as there are runnable threads in the system. Second, since the surplus α_i of a thread depends only on its start tag and not the finish tag, SFS does not require the quantum length to be known at the time of scheduling (the quantum duration q is required to compute the finish tag only after the quantum ends). This is a desirable feature since the duration of a quantum can vary if a thread blocks before it is preempted. Third, SFS ensures that blocked threads do not accumulate credit for the processor shares they do not utilize while sleeping—this is ensured by setting the start tag of a newly woken-up thread to at least the virtual time (this prevents a thread from accumulating credit by sleeping for a long duration and then starving other threads upon waking up). Finally, from the definition of α_i and the virtual time, it follows that at any instant there is always at least one thread with $\alpha_i = 0$ (this is the thread with the minimum start tag, i.e., $S_i = v$ and also has the least surplus value). Since the thread with the minimum surplus value is also the one with the minimum start tag, surplus fair scheduling reduces to start-time fair queuing (SFQ) [9] in a uniprocessor system. Thus, SFS can be viewed as a generalization of

SFQ for multiprocessor environments. We experimentally demonstrate in Section 4.3 that SFS addresses the problem of proportionate allocation in the presence of frequent arrivals and departures described in Example 2 of Section 1.2.

2.4 Fair Allocation versus Processor Affinities

SFS as defined in the previous section achieves pure fair-share allocation but does not take *processor affinities* [25] into account while making scheduling decisions. Scheduling a thread on the same processor enables it to benefit from data cached from previous scheduling instances and improves the effectiveness of a processor cache. SFS can be modified to account for processor affinities as follows. Instead of scheduling the thread with the least surplus value on a processor, SFS can instead examine the first B threads with the least surplus values and pick one which was previously scheduled on that processor. If no such thread exists, then the scheduler simply picks the thread with the least surplus value for execution. The quantity B is a tunable parameter and is referred to as the *processor affinity bias*. Using $B = 1$ reduces to pure fair-share scheduling; a large value of B increases the probability of finding a thread with an affinity for a particular processor. Observe that processor-affinity based scheduling and fair-share scheduling can be conflicting goals. Using a large processor affinity bias can cause SFS to deviate from GMS-based fair allocation but allows the scheduler to improve performance by exploiting cache locality. In contrast, a small value of the bias enables SFS to provide better fairness guarantees but can degrade cache performance.

3 Implementation Considerations

We have implemented surplus fair scheduling in the Linux kernel and have made the source code publicly available to the research community.³ The entire implementation effort took less than three weeks and was around 1500 lines of code. In the rest of this section, we present the details of our kernel implementation and explain some of our key design decisions.

3.1 SFS Data Structures and Implementation

The implementation of surplus fair scheduling was done in version 2.2.14 of the Linux kernel. Our implementation replaces the standard time sharing scheduler in Linux; the modified kernel schedules all

³The source code for our implementation is available from <http://www.cs.umass.edu/~lass/software/gms>.

threads/processes using SFS. Each thread in the system is assigned a default weight of 1; the weight assigned to a thread can be modified (or queried) using two new system calls—`setweight` and `getweight`. The parameters expected by these system calls are similar to the `setpriority` and `getpriority` system calls employed by the Linux time sharing scheduler. SFS allows the weight assigned to a thread to be modified at any time (just as the Linux time sharing scheduler allows the priority of a thread to be changed on-the-fly).

Our implementation of SFS maintains three queues. The first queue consists of all runnable threads in descending order of their weights. The other two queues consist of all runnable threads in increasing order of start tags and surplus values, respectively. The first queue is employed by the readjustment algorithm to determine the feasibility of the assigned weights (recall from Section 2.1 that maintaining a list of threads sorted by their weights enables the weight readjustment algorithm to be implemented efficiently). The second queue is employed by the scheduler to compute the virtual time; since the queue is sorted on start tags, the virtual time at any instant is simply the start tag of the thread at the head of the queue. The third queue is used to determine which thread to schedule next—maintaining threads sorted by their surplus values enables the scheduler to make scheduling decisions efficiently.

Given these data structures, the actual scheduling is performed as follows. Whenever a quantum expires or one of the currently running threads blocks, the Linux kernel invokes the SFS scheduler. The SFS scheduler first updates the finish tag of the thread relinquishing the processor and then computes its start tag (if the thread is still runnable). The scheduler then computes the new virtual time; if the virtual time changes from the previous scheduling instance, then the scheduler must update the surplus values of all runnable threads (since α_i is a function of v) and re-sort the queue. The scheduler then picks the thread with the minimum surplus and schedules it for execution. Note that since a running thread may not utilize its entire allocated quantum due to blocking events, quanta on different processors are not synchronized; hence, each processor independently invokes the SFS scheduler when its currently running thread blocks or is preempted. Finally, the readjustment algorithm is invoked every time the set of runnable threads changes (i.e., after each arrival, departure, blocking event or wakeup event), or if the user changes the weight of a thread.

3.2 Implementation Complexity and Optimizations

The implementation complexity of the SFS algorithm is as follows:

- *New arrival or a wakeup event:* The newly arrived/woken up thread must be inserted at the appropriate position in the three run queues. Since the queues are in sorted order, using a linear search for insertions takes $O(t)$, where t is the number of runnable threads. The complexity can be further reduced to $O(\log t)$ if binary search is used to determine the insert position. The readjustment algorithm is invoked after the insertion, which has a complexity of $O(p)$. Hence, the total complexity is $O(t + p)$.
- *Departure or a blocking event:* The terminated/blocked thread must be deleted from the run queue, which is $O(1)$ since our queues are doubly linked lists. The readjustment algorithm is then invoked for the new run queue, which takes $O(p)$. Hence, the total complexity is $O(p)$.
- *Scheduling decisions:* The scheduler first updates finish and start tags of the thread relinquishing the processor and computes the new virtual time, all of which are constant time operations. If the virtual time is unchanged, the scheduler only needs to pick the thread with minimum surplus (which takes $O(1)$ time). If the virtual time increases from the previous scheduling instance, then the scheduler must first update the surplus values of all runnable threads and re-sort the queue. Sorting is an $O(t \log t)$ operation, while updating surplus values takes $O(t)$. Hence, the total complexity is $O(t \log t)$. The run time performance, in the average case, can be improved by observing the following. Since the queue was in sorted order prior to the updates, in practice, the queue remains mostly in sorted order after the new surplus values are computed. Hence, we employ insertion sort to re-sort the queue, since it has good run time performance on mostly-sorted lists. Moreover, updates and sorting are required only when the virtual time changes. The virtual time is defined to be the minimum start tag in the system, and hence, in a p -processor system, typically only one of the p currently running threads have this start tag. Consequently, on average, the virtual time changes only once every p scheduling instances, which amortizes the scheduling overhead over a larger number of scheduling instances.
- *Synchronization issues:* Synchronization overheads can become an issue in SMP servers if the scheduling algorithm imposes a large overhead. Despite its $O(t \log t)$ overhead, SFS can be implemented efficiently due to the following reasons. First, we have developed a scheduling heuristic (described next) that reduces the scheduling overhead to a constant.

Second, although the readjustment algorithm needs to lock the run queue while examining the feasibility constraint for runnable threads, as explained earlier, these checks can be done efficiently in $O(p)$ time (independent of the number of threads in the system). Finally, the granularity of locks required by SFS is identical to that in the Linux SMP scheduler. In fact, our implementation reuses that portion of the code.

Since the scheduling overhead of SFS grows with the number of runnable threads, we have developed a heuristic to limit the scheduling overhead when the number of runnable threads becomes large. Our heuristic is based on the observation that $\alpha_i = \phi_i \cdot (S_i - v)$ and hence, the thread with the minimum surplus typically has either a small weight, a small start tag, or a small surplus in the previous scheduling instance. Consequently, examining a few threads with small start tags, small weights, or small prior surplus values, computing their new surpluses and choosing the thread with minimum surplus is a good heuristic in practice. Since our implementation already maintains three queues sorted by ϕ_i , S_i and α_i , this can be trivially done by examining the first few threads in each queue, computing their new surplus values and picking the thread with the least surplus. This obviates the need to update the surpluses and to re-sort every time the virtual time changes; the scheduler needs to do so only every so often and can use the heuristic between updates (infrequent updates and sorting are still required to maintain a high accuracy of the heuristic). Hence, the scheduling overhead *reduces to a constant and becomes independent of the number of runnable threads in the system* (updates to α_i and sorting continue to be $O(t \log t)$, but this overhead is amortized over a large number of scheduling instances). Moreover, since the heuristic examines multiple runnable threads, it can be easily combined with the technique proposed in Section 2.4 to account for processor affinities. We conducted several simulation experiments to determine the efficacy of this heuristic. Figure 3 plots the percentage of the time our heuristic successfully picks the thread with the minimum surplus (we omit detailed results due to space constraints). The figure shows that, in a quad-processor system, examining the first 20 threads in each queue provides sufficient accuracy ($> 99\%$) even when the number of *runnable* threads is as large as 5000 (the actual number of threads in the system is typically much larger).

As a final caveat, the Linux kernel uses only integer variables for efficiency reasons and avoids using floating point variables as a data type. Since the computation of start tags, finish tags and surplus values involves floating point operations, we simulate floating point variables using integer variables. To do so we scale each floating

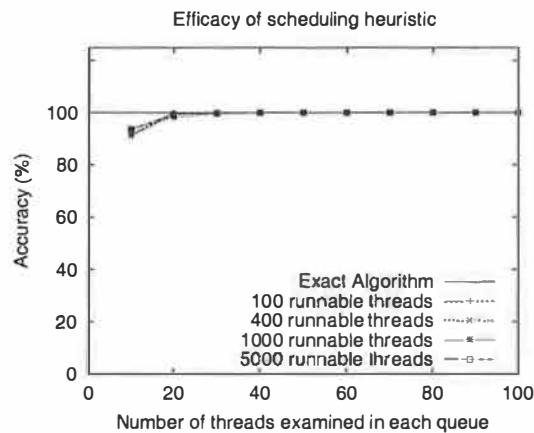


Figure 3: Efficacy of the scheduling heuristic: the figure plots the percentage of the time the heuristic successfully picks the thread with the least surplus for varying run queue lengths and varying number of threads examined.

point operation in SFS by a constant factor. Employing a scaling factor of 10^n for each floating point operation enables us to capture n places beyond the decimal point in an integer variable (e.g., the finish tag is computed as $F_i = S_i + \frac{q \cdot 10^n}{\phi_i}$). The scaling factor is a compile time parameter and can be chosen based on the desired accuracy—we found a scaling factor of 10^4 to be adequate for most purposes. Observe that a large scaling factor can hasten the wrap-around in the start and finish tags of long running threads; we deal with wraparound by adjusting all start and finish tags with respect to the minimum start tag in the system and resetting the virtual time.

4 Experimental Evaluation

In this section, we experimentally evaluate the surplus fair scheduling algorithm and demonstrate its efficacy. We conducted several experiments to (i) examine the benefits of the readjustment algorithm, (ii) demonstrate proportionate allocation of processor bandwidth in SFS, and (iii) measure the scheduling overheads imposed by SFS. We used SFQ and the Linux time sharing scheduler as the baseline for our comparisons. In what follows, we first describe the test-bed for our experiments and then present the results of our experimental evaluation.

4.1 Experimental Setup

The test-bed for our experiments consisted of a 500 MHz Pentium III-based dual-processor PC with 128 MB RAM, 13GB SCSI disk, and a 100 Mb/s 3-Com ethernet

card (model 3c595). The PC ran the default installation of Red Hat Linux 6.0. We used version 2.2.14 of the Linux kernel for our experiments; depending on the experiment, the kernel employed either SFS, SFQ or the time sharing scheduler to schedule threads. In each case, we used a quantum duration of 200 ms, which is the default quantum duration employed by the Linux kernel. The Linux kernel (and hence, our SFS scheduler) can be configured to employ finer-grain quanta; however, we do not examine the implications of doing so in this paper. All experiments were run when the system was lightly loaded. Note that due to resource constraints, our experiments were run on a system with only two processors; we have verified the efficacy of SFS on a larger number of processors via simulations (we omit these results due to space constraints).

The workload for our experiments consisted of a combination of real-world applications, benchmarks, and sample applications that we wrote to demonstrate specific features. These applications include: (i) *Inf*, a compute-intensive application that performs computations in an infinite loop; (ii) *Interact*, an I/O bound interactive application; (iii) *thttpd*, a single-threaded event-based web server, (iv) *mpeg.play*, the Berkeley software MPEG-1 decoder, (v) *gcc*, the GNU C compiler, (vi) *disksim*, a publicly-available disk simulator, (vii) *dhrystone*, a compute-intensive benchmark for measuring integer performance, and (viii) *lmbench*, a benchmark that measures various aspects of operating system performance. Next, we describe the experimental results obtained using these applications and benchmarks.

4.2 Impact of the Weight Readjustment Algorithm

To show that the weight readjustment algorithm can be combined with existing GPS-based scheduling algorithms to reduce the unfairness in their allocations, we conducted the following experiment. At $t=0$, we started two *Inf* applications (T_1 and T_2) with weights 1:10. At $t=15s$, we started a third *Inf* application (T_3) with a weight of 1. Task T_2 was then stopped at $t=30s$. We measured the processor shares received by the three applications (in terms of number of loops executed) when scheduled using SFQ; we then repeated the experiment with SFQ coupled with the weight readjustment algorithm. Observe that this experimental scenario corresponds to the infeasible weights problem described in Example 1 of Section 1.2. As expected, SFQ is unable to distinguish between feasible and infeasible weight assignments, causing task T_1 to starve upon the arrival of task T_3 at $t=15s$ (see Figure 4(a)). In contrast, when coupled with the readjustment algorithm, SFQ ensures that all tasks receive bandwidth in proportion to their instantaneous weights (1:1 from $t=0$ through $t=15$, and 1:2:1

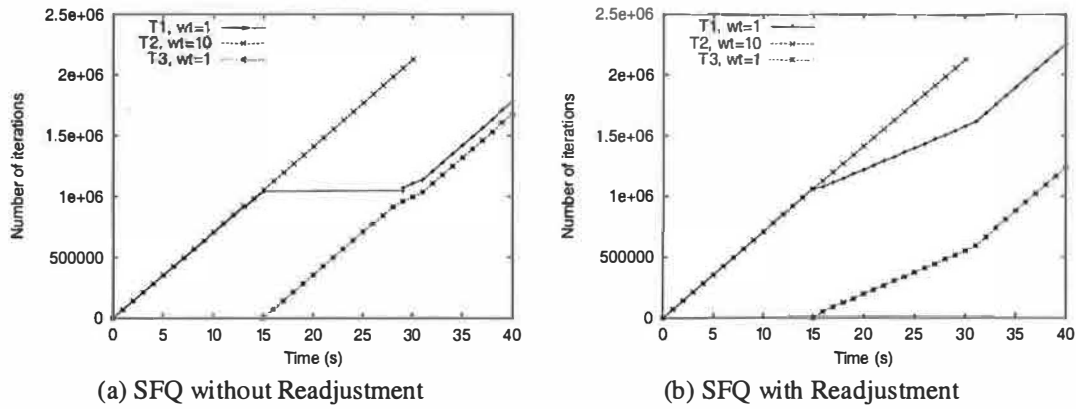


Figure 4: Impact of the weight readjustment algorithm: use of the readjustment algorithm enables SFQ to prevent starvation and reduces the unfairness in its allocations.

from $t=15$ through $t=30$, and 1:1 from then on). See Figure 4(b). This demonstrates that the weight readjustment algorithm enables a GPS-based scheduler such as SFQ to reduce the unfairness in its allocations in multiprocessor environments.

4.3 Comparing SFQ and SFS

In this section, we demonstrate that even with the weight readjustment algorithm, SFQ can show unfairness in multiprocessor environments, especially in the presence of frequent arrivals and departures (as discussed in Example 2 of Section 1.2). We also show that SFS does not suffer from this limitation. To demonstrate this behavior, we started an *Inf* application (T_1) with a weight of 20, and 20 *Inf* applications (collectively referred to as T_{2-21}), each with weight of 1. To simulate frequent arrivals and departures, we then introduced a sequence of short *Inf* tasks (T_{short}) into the system. Each of these short tasks was assigned a weight of 5 and ran for 300ms each; each short task was introduced only after the previous one finished. Observe that the weight assignment is feasible at all times, and the weight readjustment algorithm never modifies any weights. We measured the processor share received by each application (in terms of the cumulative number of loops executed). Since the weights of T_1 , T_{2-21} and T_{short} are in the ratio 20:20:5, we expect T_1 and T_{2-21} to receive an equal share of the total bandwidth and this share to be four times the bandwidth received by T_{short} . However, as shown in Figure 5(a), SFQ is unable to allocate bandwidth in these proportions (in fact, each set of tasks receives approximately an equal share of the bandwidth). SFS, on the other hand, is able to allocate bandwidth approximately in the requested proportion of 4:4:1 (see Figure 5(b)).

The primary reason for this behavior is that SFQ schedules threads in “spurts”—threads with larger

weights (and hence, smaller start tags) run continuously for some number of quanta, then threads with smaller weights run for a few quanta and the cycle repeats. In the presence of frequent arrivals and departures, scheduling in such “spurts” allows tasks with higher weights (T_1 and T_{short} in our experiment) to run almost continuously on the two processors; T_{2-21} get to run infrequently. Thus, each T_{short} task gets as much processor share as the higher weight task T_1 ; since each T_{short} task is short lived, SFQ is unable to account for the bandwidth allocated to the previous task when the next one arrives. SFS, on the other hand, schedules each application based on its surplus. Consequently, no task can run continuously and accumulate a large surplus without allowing other tasks to run first; this finer interleaving of tasks enables SFS to achieve proportionate allocation even with frequent arrivals and departures.

4.4 Proportionate Allocation and Application Isolation in SFS

Next, we demonstrate proportionate allocation and application isolation of tasks in SFS. To demonstrate proportionate allocation, we ran 20 background *dhrystone* processes, each with a weight of 1. We then ran two *thttpd* web servers and assigned them different weights (1:1, 1:2, 1:4 and 1:7). A large number of requests were then sent to each web server. In each case, we measured the average processor bandwidth allocated to each web server (the background *dhrystone* processes were necessary to ensure that all weights were feasible at all times; without these processes, no weight assignment other than 1:1 would be feasible in a dual-processor system). As shown in Figure 6(a), the processor bandwidth allocated by SFS to each web server is in proportion to its weight.

To show that SFS can isolate applications from one

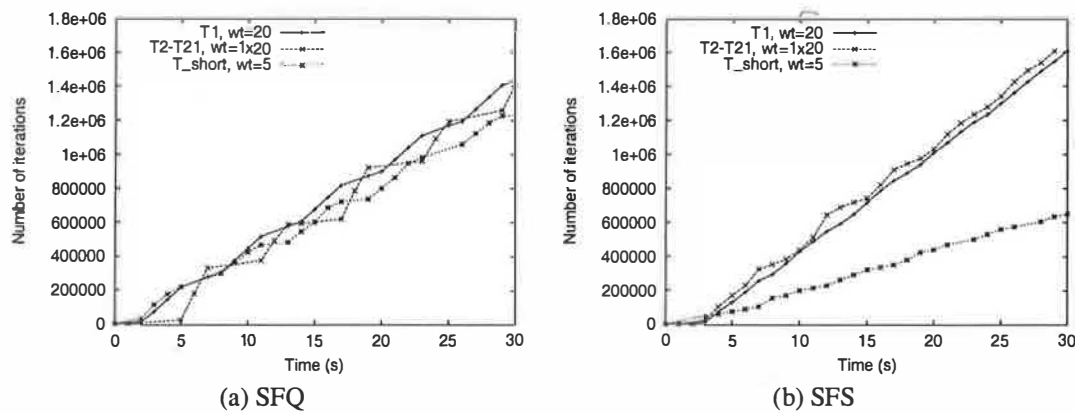


Figure 5: The Short Jobs Problem. Frequent arrivals and departures in multiprocessor environments prevent SFQ from allocating bandwidth in the requested proportions. SFS does not have this drawback.

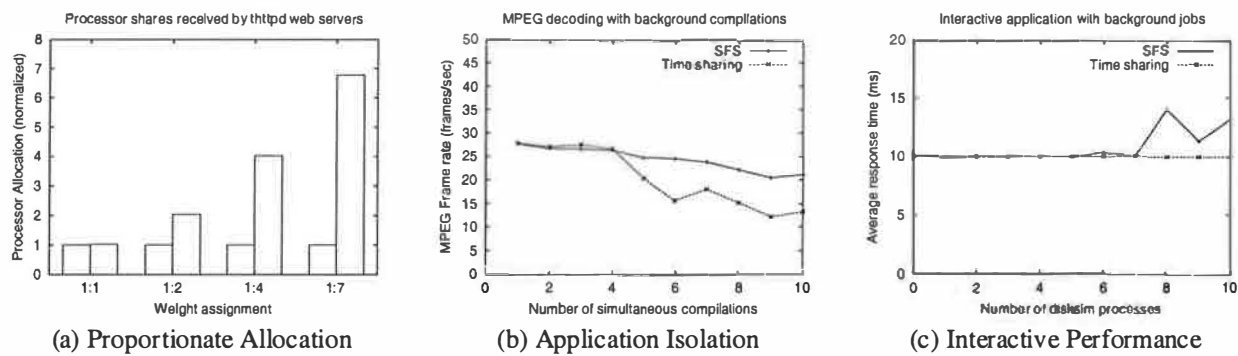


Figure 6: Proportionate allocation and application isolation in SFS. Figure (a) shows that SFS allocates bandwidth in the requested proportions. Figure (b) shows that SFS can isolate a software video decoder from background compilations. Figure (c) shows that SFS provides interactive performance comparable to time sharing

another, we ran the *mpeg_play* software decoder in the presence of a background compilation workload. The decoder was given a large weight and used to decode a 5 minute long MPEG-1 clip that had an average bit rate of 1.49 Mb/s. Simultaneously, we ran a varying number of *gcc* compile jobs, each with a weight of 1. The scenario represents video playback in the presence of background compilations; running multiple compilations simultaneously corresponds to a parallel *make* job (i.e., *make -j*) that spawns multiple independent compilations in parallel. Observe that assigning a large weight to the decoder ensures that the readjustment algorithm will effectively assign it the bandwidth of one processor, and the compilations jobs share the bandwidth of the other processor.

We varied the compilation workload and measured the frame rate achieved by the software decoder. We then repeated the experiment with the Linux time sharing scheduler. As shown in Figure 6(b), SFS is able to isolate the video decoder from the compilation work-

load, whereas the Linux time sharing scheduler causes the processor share of the decoder to drop with increasing load. We hypothesize that the slight decrease in the frame rate in SFS is caused due to the increasing number of intermediate files created and written by the *gcc* compiler, which interferes with the reading of the MPEG-1 file by the decoder.

Our final experiment consisted of an I/O-bound interactive application *Interact* that ran in the presence of a background simulation workload (represented by some number of *disksim* processes). Each application was assigned a weight of 1, and we measured the response time of *Interact* for different background loads. As shown in Figure 6(c), even in the presence of a compute-intensive workload, SFS provides response times that are comparable to the time sharing scheduler (which is designed to give higher priority to I/O-bound applications).

Test	Time sharing	SFS
syscall overhead	0.7 μ s	0.7 μ s
fork()	400 μ s	400 μ s
exec()	2 ms	2 ms
Context switch (2 proc/ 0KB)	1 μ s	4 μ s
Context switch (8 proc/ 16KB)	15 μ s	19 μ s
Context switch (16 proc/ 64KB)	178 μ s	179 μ s

Table 1: Scheduling Overheads reported by *lmbench*

4.5 Benchmarking SFS: Scheduling Overheads

We used *lmbench*, a publicly available operating system benchmark, to measure the overheads imposed by the SFS scheduler. We ran *lmbench* on a lightly loaded machine with SFS and repeated the experiment with the Linux time sharing scheduler. In each case, we averaged the statistics reported by *lmbench* over several runs to reduce experimental error. Note that the SFS code is untuned, while the time sharing scheduler has benefited from careful tuning by the Linux kernel developers. Table 1 summarizes our results (we report only those *lmbench* statistics that are relevant to the CPU scheduler). As shown in Table 1, the overhead of creating processes (measured using the `fork` and `exec` system calls) is comparable in both schedulers. The context switch overhead, however, increases from 1 μ s to 4 μ s for two 0KB processes (the size associated with a process is the size of the array manipulated by each process and has implications on processor cache performance [13]). Although the overhead imposed by SFS is higher, it is still considerably smaller than the 200 ms quantum duration employed by Linux. The context switch overheads increase in both schedulers with increasing number of processes and increasing process sizes. SFS continues to have a slightly higher overhead, but the percentage difference between the two schedulers decreases with increasing process sizes (since the restoration of the cache state becomes the dominating factor in context switches).

Figure 7 plots the context switch overhead imposed by the two schedulers for varying number of 0 KB processes (the array sizes manipulated by each process was set to zero to eliminate caching overheads from the context switch times). As shown in the figure, the context switch overhead increases sharply as the number of processes increases from 0 to 5, and then grows with the number of processes. The initial increase is due to the increased book-keeping overheads incurred with a larger number of runnable processes (scheduling decisions are trivial when there is only one runnable process and require minimal updates to kernel data structures). The increase in scheduling overhead thereafter is consistent with the complexity of SFS reported in Section 3.2 (the scheduling heuristic presented in that section was not

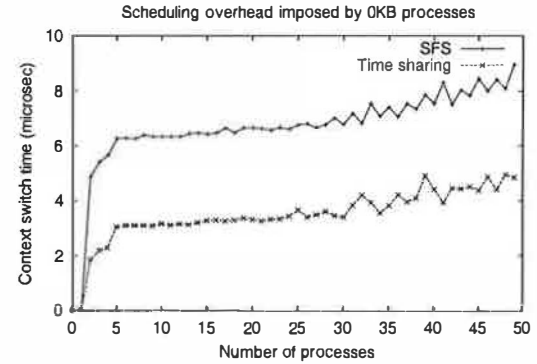


Figure 7: Scheduling overheads reported by *lmbench* with varying number of processes.

used in this experiment). Interestingly, the Linux time sharing scheduler also imposes an overhead that grows with the number of processes.

5 Limitations and Directions for Future Work

Whereas surplus fair scheduling achieves proportionate allocation of bandwidth in multiprocessor environments, it has certain limitations. In what follows, we discuss some of the limitations of SFS and opportunities for future work.

In SFS, the QoS requirements of an application are distilled to a single dimension, namely its rate (which is specified using a weight). That is, SFS is a pure proportional-share CPU scheduler. Applications can have requirements along other dimensions. For instance, interactive applications tend to be more latency-sensitive than batched applications, or a certain application may need to have higher priority than other applications. Recent research has extended GPS-based proportional-share schedulers to account for these dimensions. For instance, SMART [16] enhances a GPS-based scheduler with priorities, while BVT [7] extends a GPS-based scheduler to handle latency requirements of threads. We plan to explore similar extensions for GMS-based schedulers such as SFS as part of our future work.

GPS-based schedulers such as SFQ can perform hierarchical scheduling. This allows threads to be aggregated into classes and CPU shares to be allocated on a per-class basis. Consequently, hierarchical schedulers can handle resource principals (e.g., processes) consisting of multiple threads. Many hierarchical schedulers also support class-specific schedulers, in which the bandwidth allocated to a class is distributed among individual threads using a class-specific scheduling policy. SFS is a single-level scheduler and can only handle resource principals with a single thread. We are currently

enhancing SFS to overcome both limitations. To handle resource principals with multiple threads, we are generalizing our weight readjustment algorithm. Specifically, a resource principal with τ threads can be simultaneously scheduled on τ processors. The feasibility constraint for such a resource principal is specified as

$$\frac{w_i}{\sum_j w_j} \leq \min\left(\frac{\tau}{p}, 1\right) \quad (7)$$

We are modifying the weight readjustment algorithm to incorporate this constraint. To support hierarchical scheduling, we are modifying SFS to allow independent resource principals to be grouped into classes in a hierarchical manner. Assuming that these groups are specified in the form of a tree, our enhanced algorithm allows weights to be specified for each node (sub-class) in the tree. Our weight readjustment algorithm then ensures feasibility of the weights assigned to each node based on the number of runnable threads in that sub-tree.

SMP-based time-sharing schedulers employed by conventional operating systems take caching effects into account while scheduling threads [25]. As explained in Section 2.4, SFS can be modified to take such processor affinities into account while making scheduling decisions. However, the implications of doing so on fairness guarantees and cache performance need further investigation.

Regardless of whether resources are allocated in relative or absolute terms, a predictable scheduler will need to employ techniques to restrict the number of threads in the system in order to provide performance guarantees. While some schedulers integrate an admission control test with the scheduling algorithm, others implicitly assume that such an admission control test will be employed but do not specify a particular test. SFS falls into the latter category—the system will need to employ admission control if threads desire specific performance guarantees. Assuming such a test is employed, fair proportional-share schedulers have been shown to provide bounds on the throughput received and the latency incurred by threads [4, 9]. We are currently analyzing SFS to determine the performance guarantees that can be provided to a thread. Note, however, that the scheduling heuristic and the processor affinity bias can weaken the guarantees provided by SFS.

Finally, proportional-share schedulers such as SFS need to be combined with tools that enable a user to determine an application's resource requirements. Such tools should, for instance, allow a user to determine the processing requirements of an application (for instance, by application profiling), translate these requirements to appropriate weights, and modify weights dynamically if these resource requirements change [6, 21]. Translating application requirements such as rate to an appropriate set of weights is the subject of future research.

6 Concluding Remarks

In this paper, we argued that the infeasibility of certain weight assignments causes unfairness or starvation in many existing proportional-share schedulers when employed for multiprocessor servers. We presented a novel weight readjustment algorithm to translate infeasible weight assignments to a feasible set of weights. We showed that our algorithm enables existing proportional-share schedulers such as SFQ to significantly reduce, but not eliminate, the unfairness in their allocations. We then presented the idealized generalized multiprocessor sharing algorithm and derived surplus fair scheduling, which is a practical instantiation of GMS. We implemented SFS in the Linux kernel and demonstrated its efficacy through an experimental evaluation. Our experiments indicate that a proportional-share CPU scheduler such as SFS is not only practical but also desirable for general-purpose operating systems. As part of future work, we plan to extend SFS to do hierarchical scheduling as well as enhance proportional-share schedulers to account for priorities and delay.

Acknowledgements

We would like to thank the anonymous reviewers and our shepherd Mike Jones for their comments.

References

- [1] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster Reserves: A Mechanism for Resource Management in Cluster-based Network Servers. In *Proceedings of the ACM SIGMETRICS Conference, Santa Clara, CA*, June 2000.
- [2] G. Banga, P. Druschel, and J. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the third Symposium on Operating System Design and Implementation (OSDI'99)*, New Orleans, pages 45–58, February 1999.
- [3] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate Progress: A Notion of Fairness in Resource Allocation. *Algorithmica*, 15:600–625, 1996.
- [4] J.C.R. Bennett and H. Zhang. Hierarchical Packet Fair Queuing Algorithms. In *Proceedings of SIGCOMM'96*, pages 143–156, August 1996.
- [5] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of ACM SIGCOMM*, pages 1–12, September 1989.
- [6] J. R. Douceur and W. J. Bolosky. Progress-based Regulation of Low-importance Processes. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'99)*, Kiawah Island Resort, SC, pages 247–260, December 1999.

- [7] K. Duda and D. Cheriton. Borrowed Virtual Time (BVT) Scheduling: Supporting Latency-sensitive Threads in a General-Purpose Scheduler. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'99)*, Kiawah Island Resort, SC, pages 261–276, December 1999.
- [8] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: Resource Management using Virtual Clusters on Shared-memory Multiprocessors. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'99)*, Kiawah Island Resort, SC, pages 154–169, December 1999.
- [9] P. Goyal, X. Guo, and H.M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proceedings of Operating System Design and Implementation (OSDI'96)*, Seattle, pages 107–122, October 1996.
- [10] M B. Jones and J. Regehr. CPU Reservations and Time Constraints: Implementation Experience on Windows NT. In *Proceedings of the Third Windows NT Symposium*, Seattle, WA, July 1999.
- [11] M B. Jones, D Rosu, and M Rosu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proceedings of the sixteenth ACM symposium on Operating Systems Principles (SOSP'97)*, Saint-Malo, France, pages 198–211, December 1997.
- [12] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communication*, 14(7):1280–1297, September 1996.
- [13] L. McVoy and C. Staelin. Lmbench: Portable Tools for Performance Analysis. In *Proceedings of USENIX'96 Technical Conference*, Available from <http://www.bitmover.com/lmbench>, January 1996.
- [14] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. In *Proceedings of the IEEE ICMCS'94*, May 1994.
- [15] A. Mok and M. Dertouzos. Multiprocessor Scheduling in a Hard Real-time Environment. In *Proceedings of the Seventh Texas Conf. on Computing Systems*, November 1978.
- [16] J. Nieh and M S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proceedings of the sixteenth ACM symposium on Operating systems principles (SOSP'97)*, Saint-Malo, France, pages 184–197, December 1997.
- [17] J. Nieh and M. S. Lam. Multimedia on Multiprocessors: Where's the OS When You Really Need It? In *Proceedings of the Eighth International Workshop on Network and Operating System Support for Digital Audio and Video*, Cambridge, U.K., July 1998.
- [18] A.K. Parekh. *A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1992.
- [19] M. Shreedhar and G. Varghese. Efficient Fair Queuing Using Deficit Round Robin. In *Proceedings of ACM SIGCOMM'95*, pages 231–242, 1995.
- [20] J. Stankovic and K. Ramamritham. The Spring Kernel: A New Paradigm for Real-time Systems. *IEEE Software*, 8(3):62–73, 1991.
- [21] D C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A Feedback-driven Proportion Allocator for Real-Rate Scheduling. In *Proceedings of the third ACM Symposium on Operating systems design and implementation (OSDI'99)*, New Orleans, LA, pages 145–158, February 1999.
- [22] I. Stoica, H. Abdel-Wahab, and K. Jeffay. On the Duality between Resource Reservation and Proportional Share Resource Allocation. In *Proceedings of the ACM/SPIE Conference on Multimedia Computing and Networking (MMCN'97)*, San Jose, CA, pages 207–214, February 1997.
- [23] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and G. Plaxton. A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. In *Proceedings of Real Time Systems Symposium*, Washington, DC, pages 289–299, December 1996.
- [24] Solaris Resource Manager 1.0: Controlling System Resources Effectively. Sun Microsystems, Inc., <http://www.sun.com/software/white-papers/wp-srm/>, 1998.
- [25] R. Vaswani and J. Zahorjan. The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed Shared Memory Multiprocessors. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 26–40, October 1991.
- [26] B. Verghese, A. Gupta, and M. Rosenblum. Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors. In *Proceedings of ASPLOS-VIII*, San Jose, CA, pages 181–192, October 1998.
- [27] C. Waldspurger and W. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 1–12, November 1994.
- [28] C. Waldspurger and W. Weihl. Stride Scheduling: Deterministic Proportional-share Resource Management. Technical Report TM-528, MIT, Laboratory for Computer Science, June 1995.

Performance-Driven Processor Allocation

Julita Corbalán, Xavier Martorell, Jesús Labarta

Departament d'Arquitectura de Computadors (DAC)

Universitat Politècnica de Catalunya (UPC)

{juli,xavim,jesus}@ac.upc.es

Abstract

This work is focused on processor allocation in shared-memory multiprocessor systems, where no knowledge of the application is available when applications are submitted. We perform the processor allocation taking into account the characteristics of the application measured at run-time. We want to demonstrate the importance of an accurate performance analysis and the criteria used to distribute the processors. With this aim, we present the *SelfAnalyzer*, an approach to dynamically analyzing the performance of applications (speedup, efficiency and execution time), and the Performance-Driven Processor Allocation (PDPA), a new scheduling policy that distributes processors considering both the global conditions of the system and the particular characteristics of running applications. This work also defends the importance of the interaction between the medium-term and the long-term scheduler to control the multiprogramming level in the case of the clairvoyant scheduling policies¹. We have implemented our proposal in an SGI Origin2000 with 64 processors and we have compared its performance with that of some scheduling policies proposed so far and with the native IRIX scheduling policy. Results show that the combination of the *SelfAnalyzer*+PDPA with the medium/long-term scheduling interaction outperforms the rest of the scheduling policies evaluated. The evaluation shows that in workloads where a simple equipartition performs well, the PDPA also performs well, and in extreme workloads where all the applications have a bad performance, our proposal can achieve a speedup of 3.9 with respect to an equipartition and 11.8 with respect to the native IRIX scheduling policy.

1 Introduction

The performance of current shared-memory multiprocessors systems heavily depends on the allocation of processors to parallel applications. This is especially

important in NUMA systems, such as the SGI Origin2000 [SGI98]. This work attacks the problem of the processor allocation in an execution environment where no knowledge of the application is available when applications are submitted.

Many researchers have considered the use of application characteristics in processor scheduling [Brecht96] [Chiang94][Marsh91][Nguyen96][NguyenZV96][Parsons96]. In these works, parallel applications are characterized by different parameters such as the maximum speedup, the average parallelism, or the size of the working set. Performing the processor allocation without taking into account these characteristics can result in a bad utilization of the machine. For instance, allocating a high number of processors to a parallel application with small speedup will result in a loss of processor performance.

Traditionally, characteristics of parallel applications were calculated in two different ways. The first approach is that the user or system administrator performs several executions under different scenarios, such as the input data or the number of processors, and collects several measurements. A second approach, used in research environments [Brecht96] [Chiang94] [Helmbold90] [Leutenegger90] [Madhukar95] [Parsons96] [Sevcik94], defines a job model, characterizing the applications by a set of parameters, such as the average of parallelism or the speedup. This information is provided to the OS as an *a priori* input, to be taken into account in subsequent executions.

This approach has several drawbacks. First of all, these tests can be very time-consuming, even, they can be prohibitive due to the number of combinations. Furthermore, many times the performance of the application depends on the particular input data (data size, number of iterations). Second, the behavior of the applications is influenced by issues such as the characteristics of the processors assigned to them, or the run-time mapping of processes to processors, or the memory placement.

1. Those scheduling policies that consider the application characteristics

These issues determine the performance of the applications and are only available at run-time. Finally, the different analytic models proposed so far are not able to represent the behavior of the application at run-time. Moreover, analytic models try to characterize the application when it is individually executed, not in a shared environment. Most of the previous approaches are based on analytic models.

On the other hand, the typical way to execute a parallel application in production systems is through a long-term scheduler, i.e. a queueing system [Feitelson95]. The queueing system manages the number of applications that are executed simultaneously, usually known as the multiprogramming level¹. In this execution environment, jobs are queued until the queueing system decides to execute it. This work is based on execution environments where the applications arrival is controlled by a long-term scheduler.

This work relies on the utilization of the characteristics of the applications calculated at run-time and on using this information for processor scheduling. In particular, we propose to use the speedup and the execution time with P processors. This work is focused on demonstrating the importance of: the accuracy of the measurements of the application characteristics, the criteria used to perform the processor scheduling, and the coordination with the queueing system, in the performance that may be achieved by parallel applications. With this aim, we present: (1) a new approach to measure the speedup and the execution time of the parallel applications, the *SelfAnalyzer*, (2) a new scheduling policy that uses the speedup and the execution time to distribute processors, the *Performance-Driven Processor Allocation (PDPA)* policy, (3) and a new approach to coordinating the (medium-term) scheduler with the queueing system (long-term scheduler).

Our approach has been implemented in an Origin2000 with 64 processors. Applications from the SPECfp95 benchmark suite and from the NAS benchmarks have been used to evaluate the performance of our proposal. All the benchmarks used in the evaluation are parallelized with OpenMP [OpenMP2000] directives. Finally, in the current implementation we assume that applications are malleable [Feitelson97], applications that can adjust to changing allocations at runtime.

1. In our environment, the multiprogramming level is normally set to allow the simultaneous execution of a small number of applications (two, three or four).

Results show that the combination of the *SelfAnalyzer*+*PDPA* with the medium/long-term scheduling interaction outperforms the rest of the scheduling policies evaluated. The evaluation shows that, in workloads where a simple equipartition performs well, the *PDPA* also performs well, and in extreme workloads where all the applications have a bad performance, our proposal can achieve a speedup of 3.9 with respect to an equipartition and 11.8 with respect to the native IRIX scheduling.

The remainder of this paper is organized as follows: Section 2 presents the related work. Section 3 presents the execution environment in which we have developed this work. Section 4 presents the *PDPA* scheduling policy. Section 5 presents the evaluation of the *PDPA* compared to some scheduling policies proposed so far and the IRIX scheduling policy. Finally, section 6 presents the conclusions of this work.

2 Related Work

Many researchers have studied the use of characteristics of the applications calculated at run time to perform processor scheduling. Majumdar *et al* [Majumdar91], Parsons *et al* [Parsons96], Sevcik [Sevcik94][Sevcik89], Chiang *et al* [Chiang94] and Leutenegger *et al* [Leutenegger90] have studied the usefulness of using application characteristics in processor scheduling. They have demonstrated that parallel applications have very different characteristics such as the speedup or the average of parallelism that must be taken into account by the scheduler. All these works have been carried out using simulations, not through the execution of real applications, and assuming *a priori* information.

Some researchers propose that applications should monitor themselves and tune their parallelism, based on their performance. Voss *et al* [Voss99] propose to dynamically detect parallel loops dominated by overheads and to serialize them. Nguyen *et al* [Nguyen96] [NguyenZV96] propose *SelfTuning*, to dynamically measure the efficiency achieved in iterative parallel regions and select the best number of processors to execute them considering the efficiency. These works have demonstrated the effectiveness of using run-time information.

Other authors propose to communicate these application characteristics to the scheduler and let it to perform the processor allocation using this information. Hamidzadeh [Hamidzadeh94] proposes to dynamically optimize the processor allocation by dedicating a processor to search the optimal allocation. This proposal does not

consider characteristics of the applications, only the system performance. Nguyen *et al* [Nguyen96][NguyenZV96] also use the efficiency of the applications, calculated at run-time, to achieve an *equal-efficiency* in all the processors. Brecht *et al* [Brecht96] use parallel program characteristics in dynamic processor allocations policies, (assuming *a priori* information). McCann *et al* [McCann93] propose a scheduler that dynamically adjust the number of processors allocated to the parallel applications to improve the processor utilization. Their approach considers the application-provided idleness to allocate the processors, resulting in a large number of re-allocations.

To obtain application characteristics, previous systems have taken approaches such as the use of the hardware counters provided by the architecture, or monitoring the execution time of the different phases of the applications. Weissman [Weissman98] uses the performance counters provided by modern architectures to improve the thread locality. McCann *et al* [McCann93] monitor the idle time consumed by the processors. Nguyen *et al* [Nguyen96][NguyenZV96] combined both, the use of hardware counters and the measurement of the idle periods of the applications.

The most studied characteristic of parallel applications has been the speedup. Several theoretical studies have analyzed the relation between the speedup and other characteristics such as the efficiency. Eager, Zahorjan and Lazowska define in [Eager89] the speedup and the efficiency. Speedup is defined for each number of processors P as the ratio between the execution time with one processor and with P processors. Efficiency is defined as the average utilization of the P allocated processors. The relationship between efficiency and speedup is shown in Figure 1

$$S(P) = \frac{T(1)}{T(P)} \longrightarrow E(P) = \frac{S(P)}{P}$$

Figure 1: Speedup and efficiency definitions

Helmbold *et al* analyze in [Helmbold90] the causes of loss of speedup and demonstrate that the super-linear speedup exists basically due to memory cache effects.

Our work has several characteristics that differ from the previously mentioned proposals. First of all, with respect the parameters used by the scheduling policy, our proposal considers two characteristics of the applications: the speedup and the execution time. We also propose to consider the variation in these characteristics proportionally to the variation in the number of allo-

cated processors. Second, we differ in the way the application characteristics are acquired. We believe that parameters such as the speedup can only be accurately calculated as the relation between two measurements, as opposed to [Nguyen96]. Furthermore, since the execution time of the applications is used by the scheduler, we propose a new approach to estimate the execution time of the whole application. Our measurements are based on the time, not on the hardware performance counters. In this way our method is independent from the architecture. Third, we have implemented and evaluated our proposal using real applications and a real architecture, the Origin2000. Simulations do not consider important issues of the architecture such as the data locality. And finally, we consider the benefit provided by the interaction of the (medium-term) scheduler with the long-term scheduler (queueing system).

3 Performance-Driven Processor Allocation

This section presents the three components of this work. Figure 2 shows the general overview of our execution environment. (1) Parallel applications calculate their performance through the *SelfAnalyzer* which informs the scheduler about the achieved speedup with the current number of processors, the estimation of the execution time of the whole application, and the requested number of processors. (2) Periodically (at each *quantum*¹ expiration) the scheduler wakes up and applies the scheduling policy, the *PDPA*. The *PDPA* distributes the processors among the parallel applications considering their characteristics, global system status, such as the number of processors allocated in the previous *quantum*, and the requested number of processors of each application. Once the processor allocation has been decided, the scheduler enforces it by suspending or resuming the application's processes. The scheduler informs the applications about the number of processors assigned to each one and applications are in charge of adapting their parallelism to their current allocation. In our work, the scheduler is a user-level application, and it must enforce the processor allocation through the native operating system calls such as *suspend*, or *resume*. Finally, the scheduler interacts with the queueing system to dynamically modify the multiprogramming level (3). The result is a multiprogramming level adapted to the particular characteristics of the running applications.

1. A typical quantum value is 100 ms

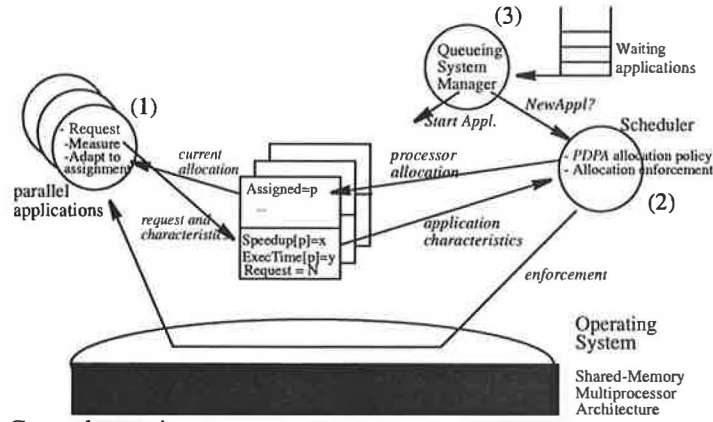


Figure 2: General overview

3.1 Dynamic Performance Analysis: *SelfAnalyzer*

The *SelfAnalyzer* [Corbalan99] is a run-time library that dynamically calculates the speedup achieved by the parallel regions, and estimates the execution time of the whole application. The *SelfAnalyzer* exploits the iterative structure of a significant number of scientific applications. The main time-consuming code of these applications is composed of a set of parallel loops inside a sequential loop. Iterations of the sequential loop have a similar behavior among them. Then, measurements for a particular iteration can be considered to predict the behavior of the next iterations, also exploited in [Nguyen96].

We believe that the speedup should be calculated as the relationship between two measurements: the sequential or reference execution time and the parallel execution time. In [Corbalan99] we demonstrated that the speedup calculated as a function of only one measurement can not detect significant issues such as the super-linear speedups. Figure 3 shows the formulation used by the *SelfAnalyzer* to calculate the speedup and to estimate the execution time.

To calculate the speedup, the *SelfAnalyzer* measures the execution time of each outer sequential iteration and also monitors the sequential and parallel regions inside the outer loop. It executes some initial iterations of the sequential loop with a predefined number of processors, (*baseline*), to be used as reference for the speedup com-

putation. Once $T(\text{baseline})$ is computed, (1) in Figure 3, the application goes on measuring the execution time but with the number of processors allocated by the scheduler. If *baseline* is one processor, the calculated speedup will correspond with the traditional speedup measurement. Since the execution of some initial iterations with one processor could consume a lot of time, we propose to set the *baseline* greater than one processor. In [Corbalan99] we demonstrate that setting *baseline* to four processors is a good trade-off between the information provided by the measurement and the amount of overhead introduced because of executing the first iterations with a small number of processors. However, this approach has the drawback that it does not allow us to directly compare speedups among applications. Setting *baseline* to four processors, the speedup with four processors of an application that scales well will be one and the speedup with four processors of an application that scales poorly will be also one.

We use Amdahl's law [Amdahl67] to normalize the speedups inside an application. Amdahl's law bounds the speedup that an application can achieve with P processors based on the fraction of sequential code.

We call this function the Amdahl's Factor (AF), see (2) in Figure 3. In this way, we calculate the AF of the *baseline* and use this value to normalize the speedups calculated by the *SelfAnalyzer*.

Considering the characteristics of these parallel applications, and taking into account their iterative structure,

$$(1) S(p) = \frac{T(\text{baseline})}{T(p)} \times AF(\text{Baseline}), \text{ where } AF(\text{Baseline}) = \frac{1}{\left(f + \frac{(1-f)}{\text{Baseline}}\right)} \quad (2)$$

$$(3) ExTime(p) = ConsumedTime + \left(\frac{AF(\text{Baseline}) \times T(\text{baseline})}{S(p)} \times ITERSRemaining\right)$$

Figure 3: Calculation of the speedup and execution time estimation

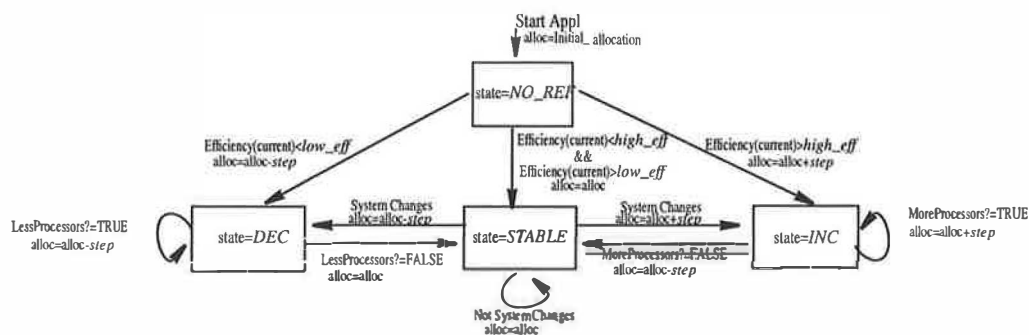


Figure 4: PDPA: Application state diagram

we are able to estimate the complete execution time of the application by using the calculated speedup and the number of iterations that the application executes, (3) in Figure 3. This estimation is calculated by adding the consumed execution time until the moment with the estimation of the remaining execution time. The remaining execution time is calculated as a function of the number of iterations not yet executed and the speedup that the application is achieving on each iteration.

To calculate the speedup and the execution time, the *SelfAnalyzer* needs to detect the following instrumentation points in the code: the starting of the application, the iterative structure, and the start and end of each parallel loop. In the current implementation, the invocation of the *SelfAnalyzer* at these points can be done in two different ways: (1) if the source code is available, the application can be re-compiled and the *SelfAnalyzer* calls can be inserted by the compiler. (2) If the source code is not available, both the iterative structure and the parallel loops are dynamically detected.

When the source code is not available, we detect the instrumentation points using dynamic interposition [Serra2000]. Calls to parallel loops are identified by the address of the function that encapsulates the loop. This sequence of values (addresses) is passed to another mechanism that dynamically detects periodic patterns. It receives as input a dynamic sequence of values and it is able to determine whether they follow a periodic pattern. Once we detect the iterative parallel region, the performance analysis is started.

In this case the number of times that the iterative structure executes is not available. In that case, the *SelfAnalyzer* is not able to estimate the execution time of the application and it assumes that the most useful characteristic to the scheduler is the execution time of one outer iteration.

As far as the status of the performance calculation is concerned, applications can be internally in two differ-

ent states: either *Performance Not yet Calculated* (PNC), or *Performance Calculated* (PC). The application is in the PNC state when the speedup with the current number of assigned processors has not been yet calculated, and in the PC when the speedup has been calculated. At the start of the application and each time the processor allocation is changed, the application is in the PNC state. If the processor allocation is modified when the application is in the PNC state, the current calculations (speedup and execution time) are discarded, and a new calculation with the current number of processors is started.

3.2 The Performance-Driven Processor Allocation: PDPA

The PDPA allocates processors among the applications considering issues such as the number of processors used in the system, the speedup achieved by each application, and the estimation of the execution time of the whole application. The goal of the PDPA is to minimize the response time, while guaranteeing that the allocated processors are achieving a good efficiency.

The PDPA considers each application to be in one of the states shown in Figure 4. These states correspond with trends of the performance of the application. These states and the transitions among them are determined both by the performance achieved by the application and by some policy parameters. The PDPA parameters are the target efficiency (*high_eff*), the minimum efficiency considered acceptable (*low_eff*), and the number of processors that will increment/decrement the application allocation (*step*). In Section 3.2.2 we will present the solution adopted in the current approach to define these parameters.

3.2.1 Application state diagram

The PDPA can assign four different states to applications: *NO_REF* (initial state), *DEC*, *INC*, and *STABLE* (see Figure 4). Each quantum the PDPA processes the performance information provided by the applications,

compared with the performance achieved in the previous quantum, and with the policy parameters, and decides the application state for next quantum. The state transitions determine the processor allocation for this application in the next quantum, even if the next state is the same.

All the applications start in the *NO_REF* state. This state means that the *PDPA* has no performance knowledge about this application (at the starting point). The processor allocation associated with the starting of a new application is the same as an equipartition (approximately $\text{total_processors_machine}/\text{total_applications}$), if there are enough free processors, otherwise it assigns the available free processors. Once the *PDPA* is informed about the achieved speedup with the previous allocation, it compares the efficiency¹ with *high_eff* and *low_eff*. If the efficiency is greater than *high_eff*, the *PDPA* considers that the application performs well and sets the next state as *INC*. If it is lower than *low_eff*, the *PDPA* considers that the application performs poorly and sets the next state as *DEC*. Finally, the *PDPA* may consider that the application has an acceptable performance that does not justify a change and the *PDPA* sets the next state as *STABLE*.

If the next state is *INC*, the application will receive in the next quantum the current number of allocated processor plus *step*. If the next state is *DEC* the application will receive in the next quantum the current number of allocated processor minus *step*. If the next state is *STABLE* the processor allocation will be maintained.

The *INC* state means that the application has performed well until the current *quantum*. In this state the *PDPA* uses both the speedup and the estimation of the execu-

tion time to decide the next state. The *MoreProcessors()* algorithm presented in Figure 5 is executed to determine the next state. *MoreProcessors()* returning *TRUE* means that the additional processors associated to the transition to this state has provided a “real benefit” to this application. In that case the next state is set to *INC*. *MoreProcessors()* returning *FALSE* means that the additional processors were not useful to the applications. In that case the next state is set to *STABLE*. If the next state is *INC*, the application will receive *step* additional processors in the next quantum. If the next state is *STABLE*, the application will loose the *step* additional processors received in the last transition.

The *DEC* state means that the application has performed badly until the current *quantum*. The *LessProcessors()* algorithm presented in Figure 5 is executed to determine the next state. *LessProcessors()* returning *TRUE* means that the application has not yet achieved an acceptable performance. In that case the next state will be *DEC*. *LessProcessors()* returning *FALSE* means that the performance is currently acceptable and the next state must be *STABLE*. If the next state is *DEC*, the application will loose *step* more processors in the next *quantum*. If the next state is *STABLE* the application will retain the current allocation.

The *STABLE* state means that the application has the maximum number of processors that the *PDPA* considers acceptable. Typically, once an application becomes *STABLE* it remains *STABLE* until it finishes. The allocation in this state is maintained. Only if the policy parameters are defined dynamically might the *PDPA* change the state of an application from *STABLE* to either *INC* or *DEC*. If *low_eff* has been increased and the efficiency achieved with the current allocation is not acceptable, the next state will be *DEC* and the application will loose *step* processors. In a symmetric way, if *high_eff* has

1. Calculated as the ratio between the speedup with *P* processors and *P*.

```

MoreProcessors()
{
    RelativeSpeedup=ExTime(LastAllocation)/ExTime(current)
    IncrementProcessors=current/LastAllocation
    if ( Efficiency(current)>=high_eff) &&
        Speedup(current)>Speedup(LastAllocation) &&
        RelativeSpeedup>=(IncrementProcessors*high_eff)) return TRUE
    else return FALSE
}
LessProcessors()
{
    if (Efficiency(current)<low_eff) return TRUE
    else return FALSE
}

```

Figure 5: Algorithms to determine if the application achieves a good or bad performance

been decreased the next state will be *INC* and the application will receive *step* additional processors.

3.2.2 PDPA parameters

As we have commented before, there are three parameters which determine the “aggressiveness” of the *PDPA*. These parameters can be either statically or dynamically defined. Statically defined, for instance by the system administrator, or dynamically defined, for instance as a function of the number of running applications.

In the current *PDPA* implementation *high_eff* and *low_eff* are dynamically defined and *step* is statically defined. The *PDPA* calculates the values of *high_eff* and *low_eff* at the start of each quantum, before processing the applications. The value of *high_eff* is calculated as a function of the ratio between the total number of processors allocated in the last *quantum* and the number of processors in the system. We have adopted this solution because this ratio is a good hint about the level of scalability that the *PDPA* must require of parallel applications to allocate them more processors. The higher this ratio is, the higher the *high_eff* value will be. Experimentally, the *high_eff* values ranges from 1.0 (ratio>0.9) to 0.7 (ratio<0.75). The value of *low_eff* is defined as a function of *high_eff*. In the current implementation it has been set to the value of *high_eff* minus 0.2.

Step is a parameter that sets the increments or decrements in the allocation of an application. This parameter is used to limit the number of re-allocations that are suffered by the applications. Setting *step* to a small value we achieve more accuracy in the number of allocated processors but the overhead introduced by the re-allocations can be significant. In the current implementation, this parameter has been tuned empirically and set to four processors.

3.2.3 Implementation issues

The *PDPA* checks the internal status of the applications and maintains the processor allocation to those applications that are in the *PNC* state. Transitions in the state diagram are only allowed either when all the applications are in the *PC* state or if there are unallocated processors. The aim of this decision is to maintain the allocation of those applications that are calculating their speedup. If we modify the speedup of an application in *PNC* state as a consequence of the processing of another application, it could result in inaccurate allocations.

To those applications that are in *PC* state, the *PDPA* allocates a minimum of one processor. This decision has been taken considering that the efficiency of an application with one processor is 1.0. This assumption is also done in scheduling policies such as the equipartition and the equal_eff. Moreover, it simplifies the *SelfAnalyzer* and the *PDPA* implementation.

Applications in *PC* state are sorted by speedup. This arrangement is done to give a certain priority to those applications that perform better, and assuring that these applications will receive processors. Finally, the *PDPA* maintains the history of the applications states, and does not allow that applications change from *STABLE* to either *DEC* or *INC* more than three times. The number of transitions is limited to avoid an excessive number of reallocations that will generate a loss of performance. It has been tuned empirically considering the particular characteristics of the workloads used. Further research with different workloads and applications will allow us to tune this parameter.

3.2.4 Interface

Table 1 shows the main primitives of the interface between the parallel library and the scheduler, and between the *SelfAnalyzer* and the scheduler. The first four rows are used by the parallel library to interact with the scheduler: requesting for cpus, checking the number

Table 1: Interface

Function	Description
int cpus_request(int P)	Request for P cpus to the scheduler
int cpus_current()	Returns the number of cpus allocated to the application
int cpus_preempted_work()	Returns the number of preempted threads
work_t get_preempted_work()	Returns the pointer to the first preempted thread
int cpus_speedup(int P, double speedup)	Sets the speedup achieved when P cpus are allocated to the application
int cpus_predicted_time(int P,double time)	Sets the execution time estimated when P cpus are allocated to the application

of allocated cpus, checking whether there are preempted threads and recovering them. These are the main functions to implement the dynamic processor allocation mechanism. The last two primitives are used by the *self-Analyzer* to inform the scheduler about the calculated speedup and the estimation of the execution time of the application.

3.3 Queueing system coordination: Dynamic multiprogramming level

As we have commented before, the multiprogramming level defines the number of applications running concurrently in the system. Non-clairvoyant scheduling policies typically allocate as many processors as possible to the running applications, since they are not able to determine how they will perform. They assign the minimum between the total requested number of processors and the number of processors of the machine.

But, even when the total requested number of processors is greater than the total number of processors in the machine, the *PDPA* may decide to leave some processors unallocated. In that case, the logical approach is to allow the queueing system to start a queued application. We propose to check after each re-allocation the scenario conditions and to decide whether a new application can be started. The conditions that must be met are the following:

- Are there free processors?
- Are all the running applications in the states *STABLE*, or *DEC*?
- Even if there are some application in the *INC* phase, does the number of unused processors reach a certain percentage? (currently defined by the administrator in a 20%)

These conditions are checked in the *NewAppl()* function call implemented by the scheduler and consulted by the queueing system.

4 Execution environment and implementation

The work done in this paper has been developed using the NANOS execution environment: The NanosCompiler, NthLib, and the CpuManager (the medium-term scheduler).

Applications are parallelized through OpenMP directives. They are compiled with the NanosCompiler [NANOS99], which generates code to NthLib [Martorell95][Martorell96]. NthLib constructs the structure of parallelism specified in the OpenMP directives and it is able to adapt the structure of the application to

the number of available processors. Moreover, it interacts with the CpuManager through a kernel interface in the following way: NthLib informs the scheduler about the number of requested processors and the scheduler informs NthLib about the number of processors available to this application.

The CpuManager [CorbalanML99] is a user-level processor scheduler. It implements the *PDPA* scheduling policy. It follows the approach proposed in [Tucker89], that assumes that applications perform better when the number of running threads is the same as the number of processors.

For the following experiments, the CpuManager implements the queueing system. Then, in this particular implementation it communicates with the *PDPA* by calling it directly. The queueing system launches a new application each time a running application finishes, and every quantum it asks to the *PDPA* whether a new application can be started.

5 Evaluation

In order to evaluate the practicality and the benefits of the *PDPA* we have executed several parallel workloads under different scenarios:

Equip: Applications are compiled with the NanosCompiler and linked with NthLib. The CpuManager is executed and it applies the equipartition policy proposed in [McCann93]. Equipartition is a space sharing policy that, to the extent possible, maintains an equal allocation of processors to all jobs. The initial allocation is set to zero. Then, the allocation number of each job is increased by one in turn, and any job whose allocation has reached the number of requested¹ processors drops out. This process continues until either there are no remaining jobs or until all *P* processors have been allocated. The only information provided by the application is its current processor requirements.

PDPA: Applications are compiled with the NanosCompiler and linked with NthLib. The CpuManager applies the *PDPA* scheduling policy. Three different variations have been executed to demonstrate the usefulness of the different components of our approach. (1) *PDPA*, as proposed in Section 3. (2) *PDPA(S)*, the *PDPA* only considers the speedup. The benefit in the execution time provided by the extra processor allocation is not considered. (3) *PDPA(idleness)*, the speedup is calculated as a

1. Specified as a command line parameter of the application or setting an environment variable

function of efficiency. In this case, we have tried to implement the approach proposed in [NguyenZV96], which calculates the efficiency measuring the sources of overhead: idleness, processor stall time, and system overhead. In our applications, we found the system overhead to be negligible, and in current architectures, like the Origin2000, the hardware does not provide the performance counters to calculate the processor stall time. Due to the difficulties of implementing their complete approach, we have implemented a similar approach only considering the idleness as source of overhead.

Equal_eff: Applications are compiled with the NanosCompiler and linked with the NthLib. The Cpu-Manager applies the equal_eff proposed in [NguyenZV96]. The goal of the equal_eff is to maximize the system efficiency. It uses the dynamically calculated efficiency of the applications, obtained through the *SelfAnalyzer*, to extrapolate [Dowdy88] the complete efficiency curve. Once extrapolated, the equal_eff works in the following way: it initially assigns a single processor to each application, and then it assigns the remaining processors one by one to the application with the currently highest (extrapolated) efficiency.

SGI-MP: Applications are compiled with the MIPSpro F77 compiler and linked with the MP-library. The commercial IRIX scheduling policy has been used. In this case, the NANOS execution environment is not involved at all. The queueing system has been used to control the multiprogramming level. In this scenario, the environment variables that define the application adaptability have been set to the following values¹:
 MP_BLOCKTIME=200000 and
 OMP_DYNAMIC=TRUE.

5.1 Architecture, applications and workloads

All the workloads have been executed in an Origin2000 [Laudon97][SGI98] with 64 processors. Each processor is a MIPS R10000 [Yeager96] at 250 MHz, with two

separated instruction and data L1 cache (32 Kbytes), and a secondary unified instruction/data cache (4 Mbytes).

To evaluate our proposal we have selected four different applications: swim, hydro2d, apsi, and BT (class=A). The swim, hydro2d and apsi are applications from the SPECfp95, the BT is from the NASPB [Jin99]. Each one of them has different behavior considering the speedup. Table 2 presents the characteristics of these applications, from higher to lower speedup. Swim achieves a super-linear speedup, BT has a moderate-high speedup, hydro2d has low speedup and apsi has very bad speedup. In all the applications, except in apsi, the maximum speedup is achieved with 32 processors. The complete performance analysis of these applications and their speedup curves can be found in [Corbalan99].

Compilation of benchmarks from the SPECfp has been done using the following command line options for the native MIPSpro f77 compiler: -64 -mips4 -r10000 -Ofast=ip27 -LNO:prefetch_ahead=1. Compilation of the BT has been done using the Makefile provided with the NASPB distribution.

Table 3 describes the four different workloads designed to evaluate the performance of the PDPA. The column instances is the number of times that the application is executed and the request column is the number of requested processors.

Workload 1 is designed to evaluate the performance of the PDPA when applications perform well, and the allocation of the equipartition policy directly achieves a good performance. Workload 2 has been designed to evaluate the PDPA performance when some of the applications perform well and some perform badly. Workload 3 evaluates the performance when applications have a medium and bad speedup and, finally, workload 4 evaluates the PDPA when all the applications have very bad performance. Since we are not assuming *a priori* knowledge of the applications, we have set the requested number of processors to 32 in all the applications.

1. These values have been tuned empirically to perform well under all the applications used in this work

Table 2: Parallel applications

Characteristic/Application(input)	swim(ref)	BT.A	hydro2d(train)	apsi(ref)
Exec.Time. in Sequential	212.2 sec.	1066.21 sec.	223.7 sec.	99 sec.
Speedup with 8/16/32 processors.	21.6/36.5/44.2	6.1/12.4/20.85	4.6/5.4/6.3	0.93/0.93/0.92

Table 3: Workload description

	swim		BT		hydro2d		apsi	
	instances	request	instances	request	instances	request	instances	request
w1	6	32	6	32				
w2			6	32			6	32
w3					6	32	6	32
w4							12	32

The multiprogramming level has been set to four in all the executions. The queueing system applies a *First Come First Served* policy, and we assume that all the applications have been queued at the same time¹.

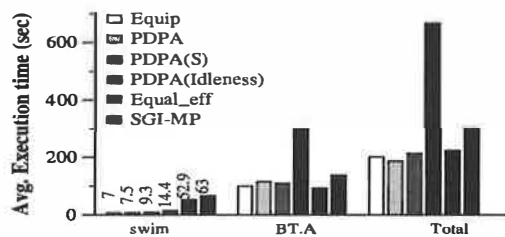
The dynamic page migration mechanism of IRIX has been activated and we have checked that results are slightly better than without this mechanism.

5.2 Results

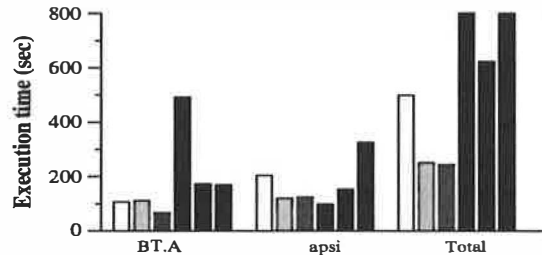
Figure 6 presents the average execution time per application in the different scenarios for the four workloads. We also show the total execution time of the workloads under the different scheduling policies. Results from workload 1 show that the *PDPA*-based scheduling policies (*PDPA* and *PDPA(S)*) perform well, compared with equipartition. The *PDPA(idleness)* does not perform well, demonstrating the importance of an accurate estimation of the performance. In this workload, the

equal_eff performs well since the applications can efficiently use a large number of processors. We can also appreciate the importance of considering the benefit provided by the additional processors to the applications. If we observe the average execution time of swim, we see how the *PDPA* outperforms the *PDPA(S)*. The reason is that the *PDPA(S)* allocates more processors to some instances of swim, allocating less processors to the rest of running applications. With the *PDPA(S)* the standard deviation in the execution time of the different instances is greater than in *PDPA*. The execution time range is (6.5,14.6) in *PDPA(S)* and (6.5,8.5) in *PDPA*. The importance of considering the benefit provided by the additional processors is more significant when the load of the system is high. In that case, without considering this parameter the processor allocation can become unfair. In the rest of workloads the difference between *PDPA* and *PDPA(S)* is less significant, since the load of the system is low.

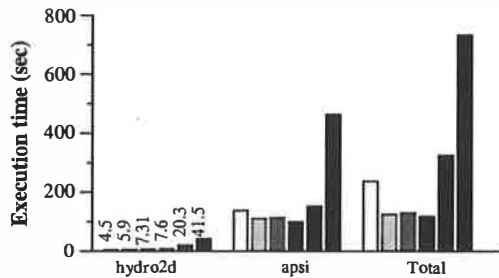
1. Instances from different applications have been merged in the queue



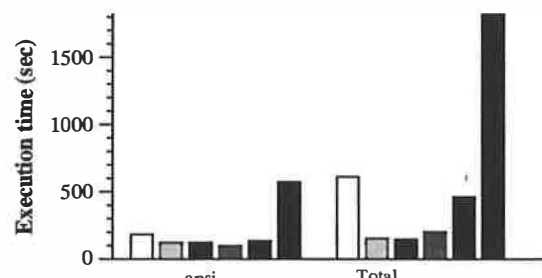
Workload 1



Workload 2



Workload 3



Workload 4

Figure 6: Per application(avg) and total execution time of the parallel workloads.

reduced because of the communication with the long-medium term scheduler. The speedup with respect to both the *PDPA*(idleness) and the SGI-MP is 3.2.

Workload 3 does not show large differences in the individual performance, although the number of processors allocated to applications by the *PDPA*-based scheduling policies is very small, allowing the long-term scheduler to start a new application, resulting in a better system utilization. This better utilization can be observed in the execution time of the workload. The *PDPA*-based scheduling policies achieve speedups from 2 (with respect to the equip.) to 6.2 (with respect to the SGI-MP).

Finally, in workload 4, the *PDPA*-based scheduling policies outperform the rest, mainly in the execution time of the workload, and also in the individual performance. Allocating a small, but sufficient, number of processors to the *apsi* avoids undesirable memory interferences. Considering the workload execution time, the *PDPA*-based scheduling policies achieve speedups from 2.0 with respect to the equip. to 6.76 with respect to the SGI-MP.

We want to comment on the performance achieved in the case of the SGI-MP environment. The problem is the large number of unnecessary context-switches. These context-switches generate a loss of performance because they imply the reload of the data cache, remote memory accesses, and increase the system time consumed by the application. For instance, consider one *apsi* execution in

the workload 4 in the *PDPA* and in the SGI-MP environments. In the *PDPA* the *apsi* has consumed a 0.1% of the execution time in system mode (0.23sec. in system-mode and 204sec. in user-mode). In the SGI-MP case, the *apsi* has spent a 27% in system mode (152sec. in system mode and 562.7sec. in user-mode).

Figure 7 shows the processor allocation made by the different scheduling policies when executing the parallel workloads. Each column shows the average of processors allocated to each different application. In these graphs, we can observe how the scheduling policies that take into account the application characteristics distribute the processors accordingly with the application performance. Since there are a minimum of two instances of each application running concurrently the highest of the columns should normally not exceed thirty-two processors (in the case of workload 4 sixteen).

We can observe how *PDPA* and *PDPA*(S) distribute the processors proportionally to the application performance. *PDPA*(S) is less restrictive and it assigns more processors. On the other hand, *equal_eff* does not have a rule to stop the processor allocation to the applications. This is the reason why the *equal_eff* allocates a higher number of processors to applications that perform badly, like *apsi*. *PDPA*(idleness) is not able to detect the good or bad behavior of the applications. The idleness is shown as a bad hint of the real efficiency achieved by the parallel applications. We can also observe in the case of the SGI-MP, how applications have adapted their par-

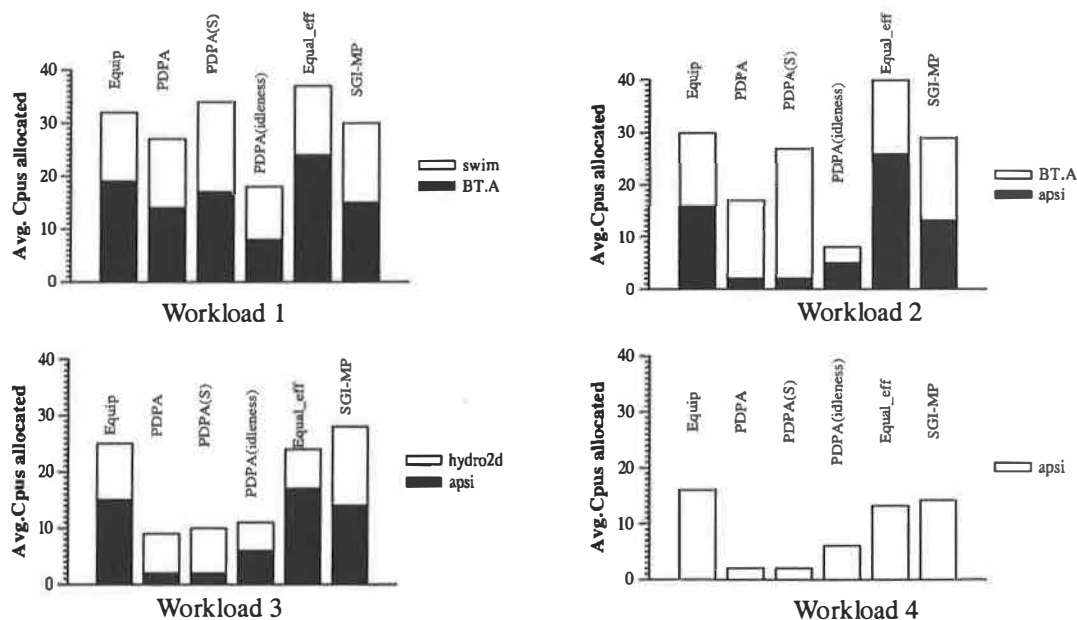


Figure 7: Processor allocation (avg) of each application under the different scheduling policies

allelism to the available processors, in a similar way to the equip.

6 Conclusions

In this work, we have presented *Performance-Driven Processor Allocation*, a new scheduling policy that uses both global system information and the application characteristics provided by the *SelfAnalyzer*, a dynamic performance analyzer. *PDPA* allocates processors to applications that will take advantage of them, avoiding unfair allocations, allocating processors to applications that do not benefit from them, or even prejudicial allocations, resulting in an increase in the execution time.

This work has been implemented and evaluated on an SGI Origin2000. We have demonstrated that it is important for the scheduler to receive accurate information about the application characteristics. Our evaluation shows that *PDPA* outperforms the considered scheduling policies.

Finally, in this work we have considered the usefulness of the interaction between the medium and the long-term scheduler. Our experience has shown that it is convenient to allow this kind of communication to improve the performance of the global system. This conclusion is valid for *PDPA* and also to any scheduling policy that allocates processors to applications based upon their performance.

7 Acknowledgments

This work has been supported by the Spanish Ministry of Education under grant CYCIT TIC98-0511, the ESPRIT Project NANOS (21907) and the Direcció General de Recerca of the Generalitat de Catalunya under grant 1999FI 00554 UPC APTIND. The research described in this work has been developed using the resources of the European Center for Parallelism of Barcelona (CEPBA).

The authors would like to thank José González and Toni Cortés for their valuable comments on a draft version of this paper.

8 References

[Amdahl67] G. M. Amdahl, "Validity of the single processor approach to achieving large-scale computing capabilities", in Proc. AFIPS, vol. 30, pp. 483-485, 1967.

[Brecht96] T. B. Brecht, K. Guha. "Using Parallel Program characteristics in dynamic processor allocation", Performance Evaluation, 27&28, pp. 519-539, 1996.

[Corbalan99] J. Corbalán, J. Labarta, "Dynamic Speedup Calculation through Self-Analysis", Technical Report number UPC-DAC-1999-43, Dep. d'Arquitectura de Computadors, UPC, 1999.

[CorbalanML99] J. Corbalán, X. Martorell, J. Labarta, "A Processor Scheduler: The CpuManager", Technical Report UPC-DAC-1999-69 Dep. d'Arquitectura de Computadors, UPC, 1999.

[Chiang94] S.-H. Chiang, R. K. Mansharamani, M. K. Vernon. "Use of Application Characteristics and Limited Preemption for Run-To-Completion Parallel Processor Scheduling Policies", In Proc. of the ACM SIGMETRICS Conference, pp. 33-44, May 1994.

[Dowdy88] L. Dowdy. "On the Partitioning of Multiprocessor Systems". Technical Report, Vanderbilt University, June 1988.

[Eager89] D. L. Eager, John Zahorjan, E. D. Lawoska. "Speedup Versus Efficiency in Parallel Systems", IEEE Trans. on Computers, Vol. 38,(3), pp. 408-423, March 1989.

[Feitelson95] D. G. Feitelson, B. Nitzberg. "Job Characteristics of a Production Parallel Scientific Workload on the NASA Ames iPSC/860", in JSSPP Springer-Verlag, Lectures Notes in Computer Science, vol. 949, pp. 337-360, 1995.

[Feitelson97] D. G. Feitelson. "Job Scheduling in Multiprogrammed Parallel Systems". IBM Research Report RC 19790 (87657), October 1994, rev. 2 1997.

[Hamidzadeh94] B. Hamidzadeh, D. J. Lilja, "Self-Adjusting Scheduling: An On-Line Optimization Technique for Locality Management and Load Balancing", Int. Conf. on Parallel Processing, vol II, pp. 39-46, 1994.

[Helmbold90] D. P. Helmbold, Ch. E. McDowell, "Modeling Speedup (n) greater than n", IEEE Transactions Parallel and Distributed Systems 1(2) pp. 250-256, April 1990.

[Jin99] H. Jin, M. Frumkin, J. Yan. "The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance". Technical Report: NAS-99-011, 1999.

- [Laudon97] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server". Proc. 24th Int. Symp. on Computer Architecture, pp. 241-251, 1997.
- [Leutenegger90] S. T. Leutenegger and M. K. Vernon. "The Performance of Multiprogrammed Multiprocessor Scheduling Policies", In Proc. of the ACM SIGMETRICS Conference, pp. 226-236, May 1990.
- [Madhukar95] M. Madhukar, J. D. Padhye, L. W. Dowdy, "Dynamically Partitioned Multiprocessor Systems", Computer Science Department, Vanderbilt University, TN 37235, 1995.
- [Majumdar91] S. Majumdar, D. L. Eager, R. B. Bunt, "Characterisation of programs for scheduling in multiprogrammed parallel systems", Performance Evaluation 13, pp. 109-130, 1991.
- [Marsh91] B. D. Marsh, T. J. LeBlanc, M. L. Scott, E. P. Markatos, "First-Class User-Level Threads". In 13th Symp. Operating Systems Principles, pp. 110-121, Oct. 1991.
- [Martorell95] X. Martorell, J. Labarta, N. Navarro and E. Ayguade, "Nano-Threads Library Design, Implementation and Evaluation". Dept. d'Arquitectura de Computadors - UPC, Technical Report: UPC-DAC-1995-33, September 1995.
- [Martorell96] X. Martorell, J. Labarta, N. Navarro and E. Ayguade, "A Library Implementation of the Nano-Threads Programming Model". Proc. of the Second Int. Euro-Par Conf., vol. 2, pp. 644-649, Lyon, France, August 1996.
- [McCann93] C. McCann, R. Vaswani, J. Zahorjan. "A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors". ACM Trans. on Computer Systems, 11(2), pp. 146-178, May 1993.
- [NANOS99] NANOS Consortium, "Nano-Threads Compiler", ESPRIT Project No 21907 (NANOS), Deliverable M3D1. Also available at <http://www.ac.upc.es/NANOS>, July 1999.
- [Nguyen96] T.D. Nguyen, J. Zahorjan, R. Vaswani, "Parallel Application Characterization for multiprocessor Scheduling Policy Design". JSSPP, vol.1162 of Lectures Notes in Computer Science. Springer-Verlag, 1996.
- [NguyenZV96] T. D. Nguyen, J. Zahorjan, R. Vaswani, "Using Runtime Measured Workload Characteristics in Parallel Processors Scheduling", in JSSPP volume 1162 of Lectures Notes in Computer Science. Springer-Verlag, 1996.
- [OpenMP2000] OpenMP Organization. "OpenMP Fortran Application Interface", v. 2.0 <http://www.openmp.org>, June 2000.
- [Parsons96] E. W. Parsons, K. C. Sevcik. "Benefits of speedup knowledge in memory-constrained multiprocessor scheduling", Performance Evaluation 27&28, pp.253-272, 1996.
- [Serra2000] A. Serra, N. Navarro, T. Cortes, "DITools: Application-level Support for Dynamic Extension and Flexible Composition", in Proceedings of the USENIX Annual Technical Conference, pp. 225-238, June 2000.
- [Sevcik94] K. C. Sevcik, "Application Scheduling and Processor Allocation in Multiprogrammed Parallel Processing Systems". Performance Evaluation 19 (1/3), pp. 107-140, Mar 1994.
- [Sevcik89] K. C. Sevcik. "Characterization of Parallelism in Applications and their Use in Scheduling". In Proc. of the ACM SIGMETRICS Conference, pp. 171-180, May 1989.
- [SGI98] Silicon Graphics Inc. Origin2000 and Onyx2 Performance Tuning and Optimization Guide. <http://techpubs.sgi.com>, Document Number 007-3430-002, 1998.
- [Tucker89] A. Tucker, A. Gupta, "Process control and scheduling issues for multiprogrammed shared-memory multiprocessors". In 12th Symposium Operating Systems Principles. pp. 159-166, December 1989.
- [Voss99] M. J. Voss, R. Eigenmann, "Reducing Parallel Overheads Through Dynamic Serialization", Proc. of the 13th Int. Parallel Processing Symposium, pp. 88-92, 1999.
- [Weissman98] B. Weissman, "Performance Counters and State Sharing Annotations: A Unified Approach to Thread Locality", Proc. of the 8th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, pp. 127 - 138, 1998.
- [Yeager96] K. C. Yeager, "The MIPS R10000 Superscalar Microprocessor". IEEE Micro vol. 16, 2 pp. 28-40, 1996.

Policies for Dynamic Clock Scheduling

Dirk Grunwald
Charles B. Morrey III

Philip Levis
Michael Neufeld

Keith I. Farkas

{grunwald,levis,cbmorrey,neufeldm}@cs.colorado.edu
Department of Computer Science
University of Colorado
Boulder, Colorado

keith.farkas@compaq.com
Compaq Computer Corp.
Western Research Lab
Palo Alto, California

Abstract

Pocket computers are beginning to emerge that provide sufficient processing capability and memory capacity to run traditional desktop applications and operating systems on them. The increasing demand placed on these systems by software is competing against the continuing trend in the design of low-power microprocessors towards increasing the amount of computation per unit of energy. Consequently, in spite of advances in low-power circuit design, the microprocessor is likely to continue to account for a significant portion of the overall power consumption of pocket computers.

This paper investigates *clock scaling* algorithms on the Itsy, an experimental pocket computer that runs a complete, functional multitasking operating system (a version of Linux 2.0.30). We implemented a number of clock scaling algorithms that are used to adjust the processor speed to reduce the power used by the processor. After testing these algorithms, we conclude that currently proposed algorithms consistently fail to achieve their goal of saving power while not causing user applications to change their interactive behavior.

1 Introduction

Dynamic clock frequency scaling and voltage scaling are two mechanisms that can reduce the power consumed by a computer. Both voltage scaling and frequency scaling are important; the power consumed by a component implemented in CMOS varies linearly with frequency and quadratically with voltage.

To evaluate the relative importance and the situations in

which either is useful, it is necessary to consider energy, the integral of power over time. By reducing the frequency at which a component operates, a specific operation will consume less power but may take longer to complete. Although reducing the frequency alone will reduce the average power used by a processor over that period of time, it may not deliver a reduction in energy consumption overall, because the power savings are linearly dependent on the increased time. While greater energy reductions can be obtained with slower clocks and lower voltages, operations take longer; this exposes a fundamental tradeoff between energy and delay.

Many systems allow the processor clock to be varied. More recently, there are a number of processors that allow the processor voltage to be changed. For example, the StrongARM SA-2 processor, currently being designed by Intel, is estimated to dissipate 500mW at 600MHz, but only 40mW when running at 150MHz – a 12-fold energy reduction for a 4-fold performance reduction [1]. Likewise, the Pentium-III processor with SpeedStep technology dissipates 9W at 500MHz but 22W at 650MHz [2], AMD has added clock and voltage scaling to the AMD Mobile K6 Plus processor family and Transmeta has also developed processors with voltage scaling. Because of this tradeoff in speed vs. power, the decision of when to change the frequency or the voltage and frequency of such processors must be made judiciously while taking into account application demand and quality of user experience.

We believe that the decision to change processor speed and voltage must be controlled by the operating system. The operating system or similar system software is the only entity with a global view of resource usage and demand. Although it is clear that the operating system should control the scheduling mechanism, it is not clear what inputs are necessary to formulate the scheduling

policy. There are two possible sources of information for policies. The application can estimate activity, providing information to the operating system about computation rates or deadlines, or the operating system can attempt to infer some policy for the applications from their behavior. These can be used separately or in concert to control voltage and processor speed.

A number of studies have investigated policies to automatically infer computation demands and adjust the processor accordingly. We have implemented those previously described algorithms; this paper describes our experience.

In the next section, we present some background material. We discuss related work in Section 3. In Section 4 we describe the schedulers we examine, our workload and our measurement methodology. We then discuss our results in Section 5.

2 Background

To better understand the importance of voltage and clock scheduling, we begin by reviewing energy-consumption concepts, then present an overview of scheduling algorithms. Lastly, we give an overview of our test platform, the Itsy Pocket Computer.

2.1 Energy

The energy E , measured in Joules (J), consumed by a computer over T seconds is equal to the integral of the instantaneous power, measured in Watts (W). The instantaneous power consumed by components implemented in CMOS, such as microprocessors and DRAM, is proportional to $V^2 \times F$, where V is the voltage supplying the component, and F is the frequency of the clock driving the component. Thus, the power consumed by a computer to, say, search an electronic phone book, may be reduced by reducing V , F , or both. However, for tasks that require a fixed amount of work, reducing the frequency may result in the system taking more time to complete the work. Thus, little or no energy will be saved. There are techniques that can result in energy savings when the processor is idle, typically through *clock gating*, which avoids powering unused devices.

In normal usage pocket computers run on batteries, which contain a limited supply of energy. However, as

discussed in [3], in practice, the amount of energy a battery can deliver (i.e., its capacity) is reduced with increased power consumption. As an illustration of this effect, consider the Itsy pocket computer that was used in this study (described in Section 2.3). When the system is idle, the integrated power manager disables the processor core but the devices remain active. If the system clock is 206 MHz, a typical pair of alkaline batteries will power the system for about 2 hours; if the system clock is set to 59 MHz, those same batteries will last for about 18 hours. Although the battery lifetime increased by a factor of 9, the processor speed was only decreased by a factor of 3.5. The capacity of the battery can also be increased by interspersing periods of high power demand with much longer periods of low power demand resulting in a “pulsed power” system [4]. The extent to which these two non-ideal properties can be exploited is highly dependent on the chemical properties and the construction of a battery as well as the conditions under which the battery is used. In general, the former effect (minimizing peak demand) is more important than the latter for the domain of pocket computers because pulsed power systems need a significant period of time to recharge the battery, and most computer applications place a more constant demand on the battery.

If a system allows the voltage to be reduced when clock speed is reduced (i.e. it supports voltage scaling), it is better to reduce the clock speed to the minimum needed rather than running at peak speed and then being idle. For example, consider a computation that normally takes 600 million instructions to complete. That application would take one second on a StrongARM SA-2 at 600MHz and would consume 500 mJoules. At 150MHz, the application would take four seconds to complete, but would only consume 160 mJoules, a four-fold savings assuming that an idle computer consumes no energy. There is obviously a significant benefit to running slower when the application can tolerate additional delay. Pering [5] used the term *voltage scheduling* to mean scheduling policies that seek to adjust both clock speed and energy. The goal of voltage scheduling is to reduce the clock speed such that all work on the processor can be completed “on time” and then reduce the voltage to the minimum needed to insure stability at that frequency.

2.2 Clock Scheduling Algorithms

In scheduling the voltage at which a system operates and the frequency at which it runs, a scheduler faces two tasks: to predict what the future system load will be (given past behavior) and to scale the voltage and clock

frequency accordingly. These two tasks are referred to as *prediction* and *speed-setting* [6]. We consider one scheduler better than another if it meets the same deadlines (or has the same behavior) as another policy but reduces the clock speed for longer periods of time.

The schedulers we implemented are *interval schedulers*, so called because the prediction and scaling tasks are performed at fixed intervals as the system runs [7]. At each interval, the processor utilization for the interval is predicted, using the utilization of the processor over one or more preceding intervals. We consider two prediction algorithms originally proposed by Weiser *et al.* [7]: PAST and AVG_N . Under PAST, the current interval is predicted to be as busy as the immediately preceding interval, while under AVG, an exponential moving average with decay N of the previous intervals is used. That is, at each interval, we compute a “weighted utilization” at time t , W_t , as a function of the utilization of the previous interval U_{t-1} and the previous weighted utilization W_{t-1} . The AVG_N policy sets $W_t = \frac{N \times W_{t-1} + U_{t-1}}{N+1}$. The PAST policy is simply the AVG_0 policy, and assumes the current interval will have the same resource demands as the previous interval.

The decision of whether to scale the clock and/or voltage is determined by a pair of boundary values used to provide hysteresis to the scheduling policy. If the utilization drops below the lower value, the clock is scaled down; similarly, if the utilization rises above the higher value, the clock is scaled up. Perring *et al.* [8] set these values at 50% and 70%. We used those values as a starting point but, as we discuss in Section 5.3, we found that the specific values are very sensitive to application behavior.

Deciding *how much* to scale the processor clock is separate from the decision of *when* to scale the clock up (or down). The SA-1100 processor used in the Itsy supports 11 different clock rates or “clock steps”. Thus, our algorithms must select one of the discrete clock steps. We use three algorithms for scaling: *one*, *double*, and *peg*. The *one* policy increments (or decrements) the clock value by one step. The *peg* policy sets the clock to the highest (or lowest) value. The *double* policy tries to double (or halve) the clock step. Since the lowest clock step on the Itsy is zero, we increment the clock index value before doubling it. Separate policies may be used for scaling upwards and downwards.

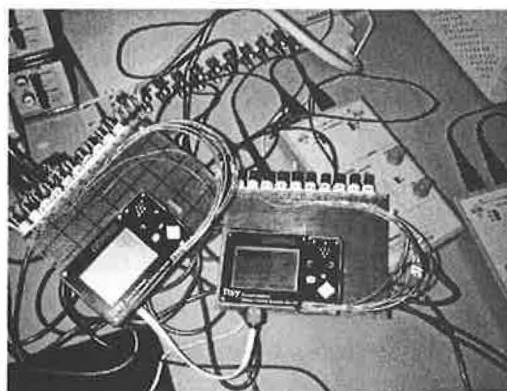


Figure 1: Equipment setups used to measure power.

2.3 The Itsy Pocket Computer

The Itsy Pocket Computer is a flexible research platform, developed to enable hardware and software research in pocket computing. It is a small, low-power, high-performance handheld device with a highly flexible interface, designed to encourage the development of innovative research projects, such as novel user interfaces, new applications, power management techniques, and hardware extensions. There are several versions of the basic Itsy design, with varying amount of RAM, flash memory and I/O devices. We used several units for this study that were modified by Compaq Computer Corporation’s Western Research Lab to include instrumentation leads for power measurement. Figure 1 shows the units along with the measurement equipment we used. We investigate the energy and power consumption of the Itsy Pocket Computer when it is run at between 59 MHz and 206 MHz, and when its StrongARM SA-1100 [9, 10] processor is powered at two different voltage levels.

All versions of the Itsy are based on the low-power StrongARM SA-1100 microprocessor. All versions have a small, high-resolution display, which offers 320×200 pixels on a 0.18mm pixel pitch, and 15 levels of greyscale. All versions also include a touchscreen, a microphone, a speaker, and serial and IrDA communication ports. The Itsy architecture can support up to 128 Mbytes both of DRAM and flash memory. The flash memory provides persistent storage for the operating system, the root file system, and other file systems and data. Finally, the Itsy also provides a “daughter card” interface that allows the base hardware to be easily extended. The Itsy uses two voltage supplies powered by the same power source. The processor core is driven by a 1.5 V supply while the peripherals are driven by a 3.3 V

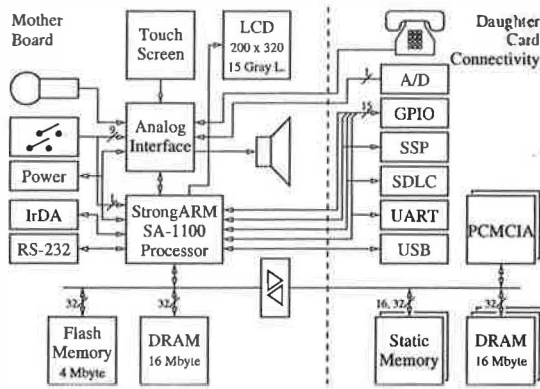


Figure 2: Itsy System Architecture

supply. Both power supplies are driven by a single 3.1V supply connected to the electrical mains.

The Itsy version 1.5 units used as the basis for this work have 64 Mbytes of DRAM and 32 Mbytes of flash memory. These units were modified to allow us to run the StrongARM SA-1100 at either 1.5 V or 1.23 V. Although 1.23 V is below the manufacturer's specification, it can be safely used at moderate clock speeds and our measurements indicate the voltage reduction yields about a 15% reduction in the power consumed by the processor; the percentage of power reduction for the system may be less than this (depending on workload) because voltage scaling only reduces the power used by the processor. The Itsy can be powered either by an external supply or by two size AAA batteries. Figure 2 shows a schematic of the Itsy architecture.

The system software of the Itsy includes a monitor and a port of version 2.0.30 of the Linux operating system. The Linux system was configured to provide support for networking, file systems and multi-user management. Applications can be developed using a number of programming environments, including C, X-Windows, SmallTalk and Java. Applications can also take advantage of available speech synthesis and speech recognition libraries.

3 Related Work

We believe that our evaluation of dynamic speed and voltage setting algorithms to be the first such empirical evaluation – to our knowledge, all previous work from different groups has relied on simulators [7, 6, 5, 11, 12]; none modeled a complete pocket computer or the work-

load likely to be run on it.

Weiser et al. [7] proposed three algorithms, OPT, FUTURE, and PAST and evaluated them using traces gathered from UNIX-based workstations running engineering applications. These algorithms use an interval-based approach that determines the clock frequency for each interval. Of the algorithms they propose, only PAST is feasible because it does not make decisions using future information that would not be available to an actual implementation. Even so, the actual version of PAST proposed by Weiser *et al.* is not implementable because it requires that the scheduler know the amount of work that had to be performed in the preceding intervals. This information was used by the scheduler to choose a clock speed that allows this delayed work to be completed in the next interval, if possible. For example, suppose post-processing of a trace revealed that the processor was busy 80% of the cycles while running at full speed. If, during re-play of the trace, the scheduler opted to run the processor at 50% speed for the interval, then 30% of the work could not be completed in that interval. Consequently, in the next interval, the scheduler would adjust the speed in an effort to at least complete the 30% “unfinished” work. Without additional information from the application, the scheduler can simply observe that the application executed until the end of the scheduling quanta, and does not know the amount of “unfinished” computing left. Because most pocket computer applications do not provide a means for the processor to know how much work should be done in a given interval, the PAST algorithm is not tractable for such systems.

The early work of Weiser et al. has been extended by several groups, including [6, 12]. Both of these groups employed the same assumptions and the same traces used by Weiser. Govil *et al.* [6] considered a large number of algorithms, while Martin [12] revised Weiser's PAST algorithm to account for the non-ideal properties of batteries and the non-linear relationship between system power and clock frequency. Martin argues that the lower bound on clock frequency should be chosen such that the number of computations per battery lifetime is maximized. While Martin correctly assumed a non-zero energy cost for idling the processor and changing clock speed, neither Govil nor Weiser did.

Both our work and that of Pering et al. [5, 11] addresses some of the limitations of the above noted earlier work. In particular, we both evaluate *implementable algorithms* using workloads that are representative of those that might be run on pocket computers. We assess the success of our algorithms under the assumption that our applications have *inelastic performance con-*

constraints and that the user should see no visible changes induced by the scheduling algorithms. By comparison, Pering *et al.* assume that frames of an MPEG video, for instance, can be dropped and present results which combine a combination of energy savings *vs.* frame rates. Our goal was to understand the performance of the different scheduling algorithms without introducing the complexity of comparing multi-dimensional performance metrics such as the percentage of dropped frames *vs.* power savings.

Pering *et al.* use intervals of 10-50ms for their scheduling calculations. In comparison to the earlier approaches presented in [7, 6, 12] in which work was considered overdue if it was not completed within an interval, both Pering *et al.* and our study consider an event to have occurred on time if delaying its completion did not adversely affect the user. However, a number of important differences exist between our work and Pering *et al.* First, Pering *et al.* model only the power consumed by the microprocessor and the memory, thus ignoring other system components whose power is not reduced by changes in clock frequency. Second, by virtue of our work using an actual implementation, we are able to evaluate longer running applications and more complex applications (e.g., Java). By virtue of their size, our applications exhibit more significant memory behavior, and thus, expose the non-linear relationship between power and clock speed noted by Martin. Lastly, by using an actual system, our scheduling implementations were exposed to periodic behaviors that are captured by traces; for example, the Java implementation uses a 30ms polling loop to check for I/O events. This periodic polling adds additional variation to the clock setting algorithms, inducing the sort of instability we will explain in §5.3.

4 Methodology

Before describing the implementation of the clock and voltage scheduling algorithms we used, it is important to understand how we did our measurements. Section 4.1 describes how we measure power and energy. We then describe the implementation of the schedulers and the workloads we used to assess their performance.

4.1 Measuring Power and Total Energy

To measure the instantaneous power consumed by the Itsy, we use a data acquisition (DAQ) system to record the current drawn by the Itsy as it is connected to an external voltage supply, and the voltage provided by this supply. Figure 1 presents a picture of our setup along with the wires connected to the Itsy to facilitate measuring the supply current¹ and voltage. We configured the DAQ system to read the voltage 5000 times per second, and convert these readings to 16-bit values. These values were then forwarded to a host computer, which stored them for subsequent analysis. From these measurements, we can compute a time profile of the power used by an application as it runs on the Itsy.

To determine the relevant part of the power-usage profile of a workload, we measure the time required to execute the workload and then select the relevant set of measurements from the data collected by the DAQ system. For each benchmark, we used the `gettimeofday` system call to time its execution; this interface uses the 3.6 MHz clock available on the processor to provide accurate timing information. To synchronize the collection of the voltages with the start of execution of a workload, as the workload begins executing, we toggle one of the SA1100's general-purpose input-output (GPIO) pins. This pin is connected to the external trigger of the DAQ system; toggling the GPIO causes the DAQ system to begin recording measurements. As our measurement technique is very similar to that which we used in [13], we refer the reader to this reference for a more in-depth description.

Once the relevant part of the profile has been determined, we use it to calculate the average power and the total energy consumed by the Itsy during the corresponding time interval. To compute the energy, we make the assumption that the power measured at time t represents the average power of the Itsy for the interval t to $t + 0.0002$ seconds, where 0.0002 seconds is the time between each successive power measurement. Thus, the energy E is equal to $\sum_{i=1}^n p_i(t) \times 0.0002$, where $p_1(t), \dots, p_n(t)$ are the n power readings of interest.

In making our power measurements, we used a similar approach as the one used in [13] to reduce a number of sources of possible measurement error. We mea-

¹The supply current was measured by measuring the voltage drop across a high precision small-valued resistor of a known resistance (0.02Ω). The current was then calculated by dividing the voltage by the resistance.

sured multiple runs of each workload; in general, we found the 95% confidence interval of the energy to be less than 0.7% of the mean energy. This implies that the runs were very repeatable, despite the possible variation that would arise from interactions between application threads, other processes and system daemons.

4.2 Workload

We used a varied workload to assess the performance of the different clock scaling algorithms. Since it's not clear what applications will be common on pocket computers, we used some obvious applications (web browsing, text reading) and other less obvious applications (chess, mpeg video and audio). The applications ran either directly on top of the Linux operating system or within a Java virtual machine [14]. To capture repeatable behavior for the interactive applications, we used a tracing mechanism that recorded timestamped input events and then allowed us to replay those events with millisecond accuracy. We did not trace the mpeg playback because there is no user interaction, and we found little inter-run variance. We used the following applications:

MPEG: We played a 320x200 color MPEG-1 video and audio clip at 15 frames a second. The mpeg video was rendered as a greyscale image on the Itsy. Audio was rendered by sending the audio stream as a WAV file to an audio player which ran as a separate process, forked from the video player. There is no explicit synchronization between the audio and video sequences, but both are sequenced to remain synchronized at 15 frames/second. The clip is 14 seconds and was played in a loop to provide 60 seconds of playback.

Web: We used a Javabeau version of the IceWeb browser to view content stored on the itsy. We selected a file containing a stored article from `www.news.com` concerning the Itsy. We scrolled down the page, reading the full article. We then went back to the root menu and opened a file containing an HTML version of WRL technical report TN-56, which has many tables describing characteristics of power usage in Itsy components. The overall trace was 190 seconds of activity.

Chess: We used a Java interface to version 16.10 of the Crafty chess playing program. Crafty was run as

a separate process. Crafty uses a play book for opening moves and then plays for specific periods of time in later stages of the games and plays the best move available when time expires. The 218 second trace includes a complete game of Crafty playing against a novice player (who lost, badly).

TalkingEditor: We used a version of the "mpedit" Java text editor that had been modified to read text files aloud using the DECtalk speech synthesis system (which is run in a separate process). The input trace records the user selecting a file to be opened using the file dialogue, (*i.e.* moving to the directory of the short text file and selecting the file), then having it spoken aloud and finally opening and having another text file read aloud. The trace took 70 seconds.

The Kaffe Java system [14] uses a JIT, makes extensive use of dynamic shared libraries and supports a threading model using `setjmp/longjmp`. The graphics library used by Java is a modified version of the publically available GRX graphics library and uses a polling I/O model to check for new input every 30 milliseconds. The MPEG player renders directly to the display.

4.3 Implementing the Scheduling Algorithms

We made two modifications to the Linux kernel to support our clock scheduling algorithms and data recording. The first modification provides a log of the process scheduler activity. This component is implemented as a kernel module with small code modifications to the scheduler that allow the logging to be turned on and off. For each scheduling decision, we record the process identifier of the process being scheduled, the time at which it was scheduled (with microsecond resolution) and the current clock rate.

We also implemented an extensible clock scaling policy module as a kernel module. We modified the clock interrupt handler to call the clock scheduling mechanism if it has been installed, and the Linux scheduler to keep track of CPU utilization. In Linux, the idle process always uses the zero process identifier. The idle process enters a low-power "nap" mode that stalls the processor pipeline until the next scheduling interval. If the previous process was not the idle process, the kernel adds the execution time to a running total. On every clock interrupt, this total is examined by the clock scaling module and then cleared. The CPU utilization can be calculated by comparing the time spent non-idle to the time length

of a quantum. Our time quantum was set to 10 msec, the default scheduling period in Linux; Pering et al. [5, 11] used similar values for their calculations.

Normally, a process can run for several quanta before the scheduler is called. The executing process is interrupted by the 100Hz system clock when the O/S decrements and examines a counter in the process control block at each interrupt. When that counter is zero, the scheduler is called. We set the counter to one each time we schedule a process, forcing the scheduler to be called every 10ms. While this modification adds overhead to the execution of an application, it allows us to control the clock scaling more rapidly. We measured the execution overhead and found it to be very small (about 6 microseconds for each 10ms interval, or 0.06%).

5 Results

The purpose of our study is to determine if the heuristics developed in prior studies can be practically applied to actual pocket computers. We examined a number of policies, most of which are variants of the AVG_N policy. As described in §4.3, we used three different speed setting policies. Our intent was to focus on systems that could be implemented in an actual O/S and that did not require modifications to the applications (such as requiring information about deadlines or schedules). We assumed that our workloads had *inelastic constraints*; in other words, we assumed the applications had no way to accommodate “missed deadlines”.

We split the discussion of our results into three parts. The first section describes aspects of the applications and how they differ from those used in prior work and the second section discusses the performance of the different clock scheduling algorithms. Finally, we examine the benefit of the limited voltage scaling available on the Itsy and summarize the results.

5.1 Application Characteristics

Figure 3 presents plots of the processor utilization over time for each of the benchmark applications. This information was gathered using the on-line process logging facility that we added to the kernel. Due to kernel memory limitations, we could only capture a subset of the process behavior. Each application was able to run at 132MHz and still meet any user interaction constraints

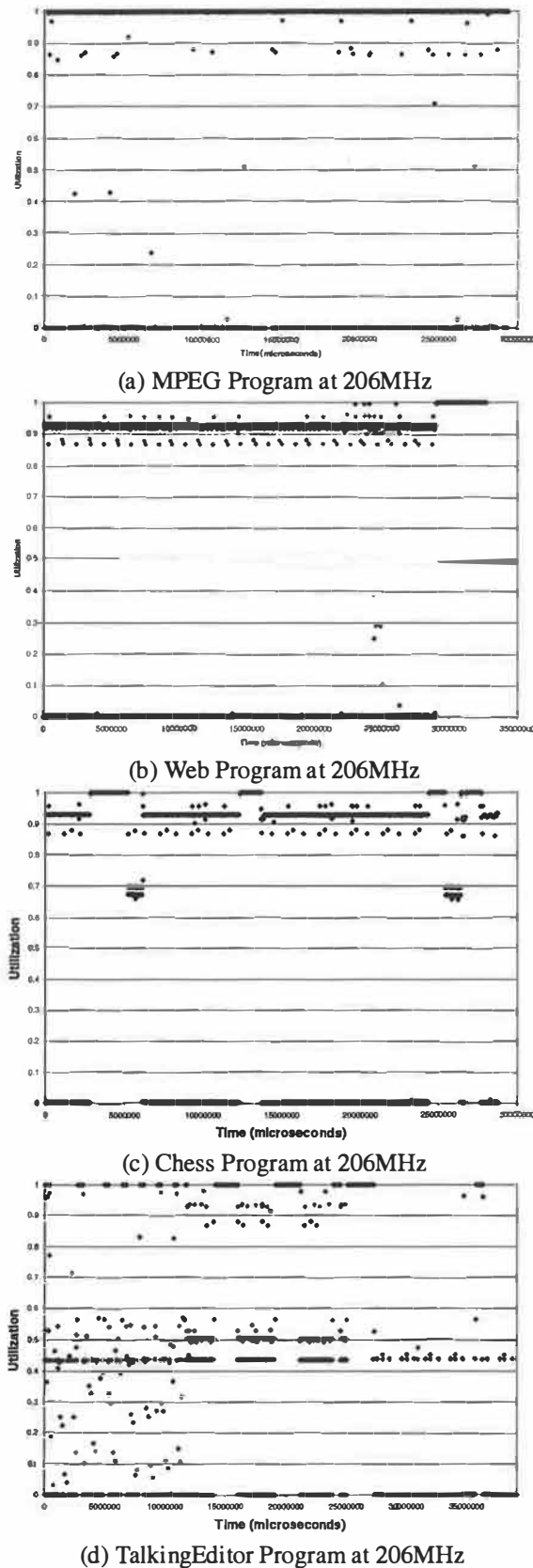


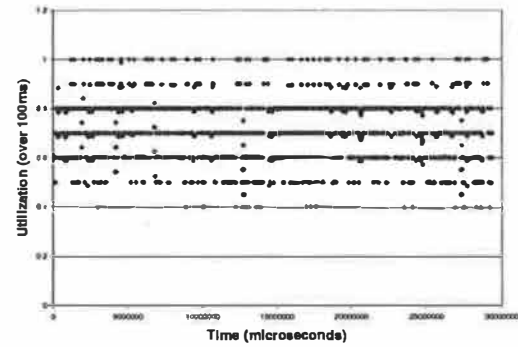
Figure 3: Utilization using 10ms Moving Average For Between 30 to 40 Second Intervals Using 206MHz Frequency Setting

(i.e. the application did not appear to behave any differently).

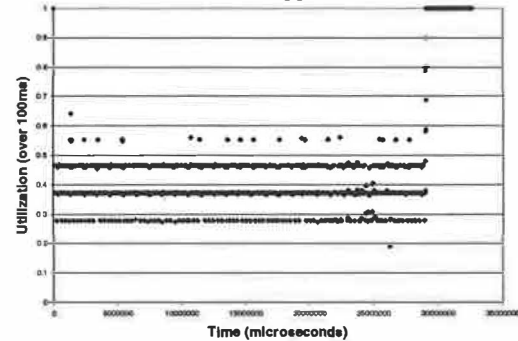
The utilization is computed for each 10ms scheduling quantum. We used the same 10ms interval for logging that is used for scheduling within Linux. Since most processes compute for several quanta before yielding, the system is usually either completely idle or completely busy during a given quantum. Some processes execute for only a short time then yield the processor prior to the end of their scheduling quanta; for example, the Java implementation we used has a 30ms I/O polling loop – thus, when the Java system is “idle,” there is a constant polling action every 30ms that takes about a millisecond to complete.

The behavior of the applications is difficult to predict, even for applications that should have very predictable behavior and each application appears to run at a different time-scale. The MPEG application renders at 15 frames/sec; there are 450 frames in the 30 second interval shown in Figure 3. Each frame is rendered in 67ms or just under 7 scheduling quanta. Any scheduling mechanism attempting to use information from a single frame (as opposed to a single quanta) would need to examine at least 7 quanta. Other applications have much coarser behavior. For example, the TalkingEditor application consumes varying amount of CPU time until the text is being loaded for speech synthesis. The bursty behavior prior to the speech synthesis results from dragging images, JIT’ing applications and opening files. Following this are long bursts of computation as the text is actually synthesized and send to the OSS-compatible sound driver. Finally, more cycles are taken by the sound driver. Thus, this application is bursty at a higher level.

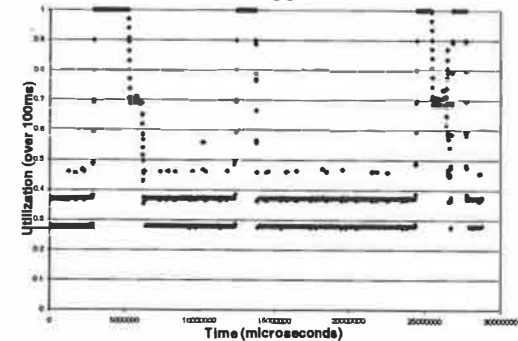
For most applications, patterns in the utilization are easier to see if you plot the utilization using a 100ms moving average, as shown in Figure 4. The MPEG application, in Figure 4(a), is still very sporadic because of inter-frame variation; for MPEG, there is even significant variance in CPU utilization (60-80%) when considering a 1 second moving average (not shown). The Chess and TalkingEditor applications show patterns influenced by user interaction. It’s clear from Figure 4(c) that utilization is low when the user is thinking or making a move and that utilization reaches 100% when Crafty is planning moves. Likewise, Figure 4(d) shows the aforementioned pattern of synthesis and sound rendering.



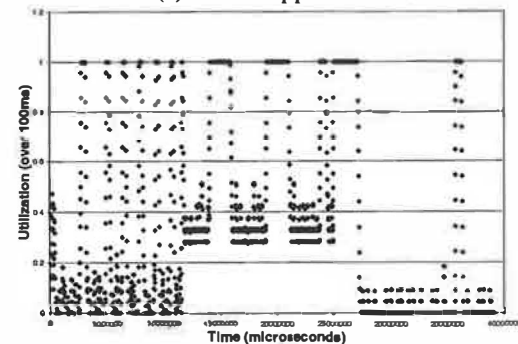
(a) MPEG Application



(b) Web Application



(c) Chess Application



(d) TalkingEditor Application

Figure 4: Utilization using 100ms Moving Average For Between 30 to 40 Second Intervals Using 206MHz Frequency Setting

5.2 Clock Scheduling Comparison

The goal of a clock scheduling algorithm is to try to predict or recognize a CPU usage pattern and then set the CPU clock speed sufficiently high to meet the (predicted) needs of that application. Although patterns in the utilization are more evident when using a 100ms sliding average for utilization, we found that averaging over such a long period of time caused us to miss our “deadline”. In other words, the MPEG audio and video became unsynchronized and some other applications such as the speech synthesis engine had noticeable delays. This occurs because it takes longer for the system to realize it is becoming busy.

This delay is the reason that the studies of Govil *et al.* [6] and Weiser [7] argued that clock adjustment should examine a 10-50ms interval when predicting future speed settings. However, as Figure 3 shows, it is difficult to find any discernible pattern at the smaller time-scales. Like Govil *et al.*, we also allowed speed setting to occur at any interval; Weiser *et al.* did not model having the scheduler interrupted while an application was running, but rather deferred clock speed changes to occur only when a process yielded or began executing in a quanta.

There are a number of possible speed-setting heuristics we could examine; since we were focusing on *implementable* policies, we primarily used the policies explored by Pering *et al.* [5]. We also explored other alternatives. One simple policy would determine the number of “busy” instructions during the previous N 10ms scheduling quanta and predict that activity in the next quanta would have the same percentage of busy cycles. The clock speed would then be set to insure enough busy cycles.

This policy sounds simple, but it results in exceptionally poor responsiveness, as illustrated in Figure 5. Figure 5(a) shows the speed changes that would occur when the application is moving from period of high CPU utilization to one of low utilization; the speed changes to 59MHz relatively quickly because we are adding in a large number of idle cycles each quanta. By comparison, when the application moves from an idle period to a fully utilized period, the simple speed setting policy makes very slow changes to the processor utilization and thus the processor speed increases very slowly. This occurs because the total number of non-idle instructions across the four scheduling intervals grows very slowly.

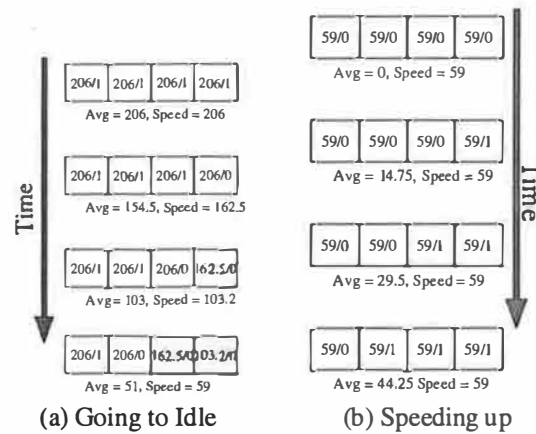


Figure 5: Simple averaging behavior results in poor policies. Each box represents a single scheduling interval, and the scheduling policy averages the number of non-idle instructions over the four scheduling quanta to select the minimum processor speed. To simplify the example, we assume each interval is either fully utilized or idle. The notation “206/0” means the CPU is set to 206MHz and the quanta is idle while “206/1” means the CPU is fully utilized.

5.3 The AVG_N Scheduler

We had initially thought that a policy targeting the necessary number of non-idle cycles would result in good behavior, but the previous example highlights why we use the speed-setting policies described in §4.3. We used the same AVG_N scheduler proposed by Govil [6] and Pering [5] and also examined by Pering *et al.* in [5]; Perings later paper in [11] did not examine scheduler heuristics and only used real-time scheduling with application-specified scheduling goals.

Our findings indicate that the AVG_N algorithm can not settle on the clock speed that maximizes CPU utilization. Although a given set of parameters can result in optimal performance for a single application, these tuned parameters will probably not work for other applications, or even the same application with different input. The variance inherent in many deadline-based applications prevents an accurate assessment of the computational needs of an application. The AVG_N policy can be easily designed to ensure that very few deadlines will be missed, but this results in minimal energy savings. We use an MPEG player as a running example in this section, as it best exemplifies behavior that illustrates the multitude of problems in past-based interval algorithms. Our intuition is that if there’s a single application that illustrates simple, easy-to-predict behavior, it should be

MPEG. Our measurements showed that the MPEG application can run at 132MHz without dropping frames and still maintain synchronization between the audio and video. An ideal clock scheduling policy would therefore target a speed of 132MHz.

However, without information from the user level application, a kernel cannot accurately determine what deadlines an application operates under. First, an application may have different deadline requirements depending on its input; for example, an MPEG player displaying a movie at 30fps has a shorter deadline than one running at 15fps. Although the deadlines for an application with a given input may be regular, the computation required in each deadline interval can vary widely. Again, MPEG players demonstrate this behavior; I-frames (key or reference) require much more computation than P-frames (predicted), and do not necessarily occur at predictable intervals.

One method of dealing with this variance is to look at lengthy intervals which will, by averaging, reduce the variance of the computational observations. Our utilization plots showed that even using 100ms intervals, significant variance is exhibited. In addition to interval length, the number of intervals over which we average (N) of the AVG_N policy can also be manipulated. We conducted a comprehensive study and varied the value of N from 0 (the PAST policy) to 10 with each combination of the speed-setting policies (*i.e.* using “peg” to set the CPU speed to the highest point, or “one” to increment or decrement the speed).

Our conclusions from the results with our benchmarks is that the weighted average has undesirable behavior. The number of intervals not only represents the length of interval to be considered; it also represents the lag before the system responds, much like the simple averaging example described above. Unlike that simple policy, once AVG_N starts responding, it will do so quickly. For example, consider a system using an AVG_9 mechanism with an upper boundary of 70% utilization and “one” as the algorithms used to increment or decrement the clock speed. Starting from an idle state, the clock will not scale to 206MHz for 120 ms (12 quanta). Once it scales up, the system will continue to do so (as the average utilization will remain above 70%) unless the next quantum is partially idle. This occurs because the previous history is still considered with equal weight even when the system is running at a new clock value.

The boundary conditions used by Pering in [5] result in a system that scales more rapidly down than up. Table 1 illustrates how this occurs. If the weighted average is

Time(ms)	Idle/Active	AVG<9>	Notes
10	Active	1000	
20	Active	1900	
30	Active	2710	
40	Active	3439	
50	Active	4095	
60	Active	4685	
70	Active	5217	
80	Active	5965	
90	Active	6125	
100	Active	6513	
110	Active	6861	
120	Active	7175	Scale up
130	Active	7458	Scale up
140	Active	7712	Scale up
150	Active	7941	Scale up
160	Idle	7146	Scale up
170	Idle	6432	
180	Idle	5789	
190	Idle	5210	
200	Idle	4689	Scale down

Table 1: Scheduling Actions for the AVG_9 Policy

70%, a fully active quantum will only increase the average to 73% while a fully idle quantum will reduce it to 63% – thus, there is a tendency to reduce the processor speed.

The job of the scheduler is made even more difficult by applications that attempt to make their own scheduling decisions. For example, the default MPEG player in the Itsy software distribution uses a heuristic to decide whether it should sleep before computing the next frame. If the rendering of a frame completes and the time until that frame is needed is less than 12ms, the player enters a spin loop; if it is greater than 12ms, the player relinquishes the processor by sleeping. Therefore, if the player is well ahead of schedule, it will show significant idle times; once the clock is scaled close to the optimal value to complete the necessary work, the work seemingly increases. The kernel has no method of determining that this is wasteful work.

Furthermore, there is some mathematical justification for our assertion that AVG_N fundamentally exhibits undesirable behavior, and will not stabilize on an optimal clock speed, even for simple and predictable workloads. Our analysis only examines the “smoothing” portion of AVG_N , not the clock setting policy. Nevertheless, it works well enough to highlight the instability issues with AVG_N by showing that, even if the system is started out at the ideal clock speed, AVG_N smoothing will still result in undesirable oscillation.

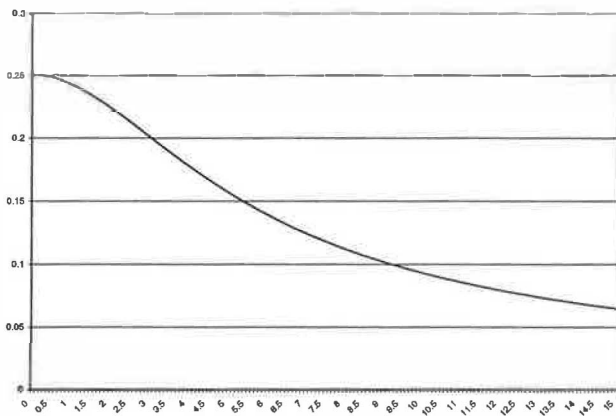


Figure 6: Fourier Transform of a Decaying Exponential

A processor workload over time may be treated as a mathematical function, taking on a value of 1 when the processor is busy, and 0 when idling. Borrowing techniques from signal processing allows us to characterize the effect of AVG_N on workloads in general as well as specific instances. AVG_N filters its input using a decaying exponential weighting function. For our implementation, we used a recursive definition in terms of both the previous actual (U_{t-1}) and weighted (W_{t-1}) utilizations: $W_t = \frac{N \times W_{t-1} + U_{t-1}}{N+1}$. For the analysis, however, it is useful to transform this into a less computationally practical representation, purely in terms of earlier unweighted utilizations. By recursively expanding the W_{t-1} term and performing a bit of algebra, this representation emerges: $W_t = \frac{1}{N+1} \sum_{k=0}^{t-1} \left(\frac{N}{N+1}\right)^{k-(t-1)} U_k$. This equation explicitly shows the dependency of each W_t on all previous U_t , and makes it more evident that the weighted output may also be expressed as the result of discretely convolving a decaying exponential function with the raw input. This allows us to examine specific types of workloads by artificially generating a representative workload and then numerically convolving the weighting function with it. We can also get a qualitative feel for the general effects AVG_N has by moving to continuous space and looking at the Fourier transform of a decaying exponential, since convolving two functions in the time domain is equivalent to multiplying their corresponding Fourier transforms.

Lets begin by examining the Fourier transform of a decaying exponential: $x(t) = e^{-\alpha t}u(t)$, where $u(t)$ is the unit step function, 0 for all $t < 0$ and 1 for $t \geq 0$. This captures the general shape of the AVG_N weighting function, shown in Figure 6. Its Fourier transform is $X(\omega) = \frac{1}{i\omega + \alpha}$. The transform attenuates, but does not eliminate, higher frequency elements. If the input signal oscillates, the output will oscillate as well. As α gets

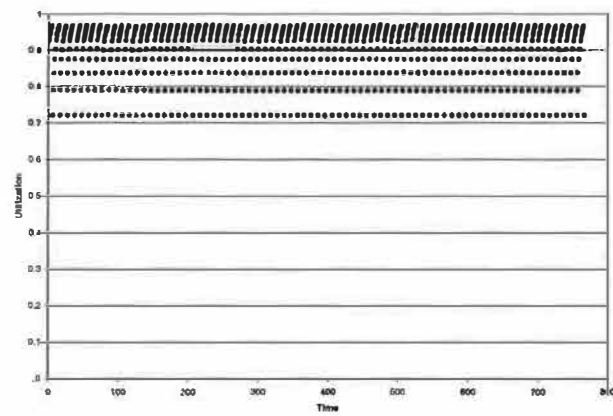


Figure 7: Result of AVG_3 Filtering on a the Processor Utilization for a Periodic Workload Over Time

smaller the higher frequencies are attenuated to a greater degree, but this corresponds to picking a larger value for N in AVG_N and comes at the expense of greater lag in response to changing processor load.

For a specific workload example, we'll use a simple repeating rectangle wave, busy for 9 cycles, and then idle for 1 cycle. This is an idealized version of our MPEG player running roughly at an optimal speed, i.e. just idle enough to indicate that the system isn't saturated. Ideally, a policy should be stable when it has the system running at an optimal speed. This implies that the weighted utilization should remain in a range that would prevent the processor speed from changing. However, as was fore-shadowed by our initial qualitative discussion, this is not the case. A rectangular wave has many high frequency components, and these result in a processor utilization as shown in Figure 7. This figure shows the oscillation for this example, and shows that oscillation occurs over a surprisingly wide range of the processor utilization. As discussed earlier, our experimental results with the MPEG player on the Itsy also exhibit this oscillation because that application exhibits the same step-function resource demands exhibited by our example.

We also simulated interval-based averaging policies that used a pure average rather than an exponentially decaying weighting function, but our simulations indicated that that policy would perform no better than the weighted averaging policy. Simple averaging suffers from the same problems experienced by the weighted averaging if you do not average the appropriate period.

5.4 Summary of Results

We are omitting a detailed exposition on the scheduling behavior of each scheduling policy primarily because most of them resulted in equivalent (and poor) behavior. Recall that the best possible scheduling goal for MPEG would be to switch to a 132MHz speed and continue to render all the frames at that speed. **No heuristic policy that we examined achieved this goal.** Figure 8 shows the clock setting behavior of the best policy we found. That policy uses the PAST heuristic (*i.e.* AVG_0) and “pegs” the CPU speed either to 206MHz or 59MHz depending on the weight metric. The bounds on the hysteresis where that a CPU utilization greater than 98% would cause the CPU to increase the clock speed and a CPU utilization less than 93% would decrease the clock speed.

This policy is “best” because it never misses any deadline (across all the applications) and it also saves a small but significant amount of energy. This last point is illustrated in Table 2. This table shows the 95% confidence interval for the average energy needed to run the MPEG application. The reduction in energy between 206MHz and 132MHz occurs because the application wastes fewer cycles in the application idle loop used to meet the frame delays for the MPEG clip. A $\approx 8\%$ energy reduction occurs when we drop the processor voltage to 1.23V – this is less than the 15% maximum reduction we measured because the application uses resources (*e.g.* audio) that are not affected by voltage scaling.

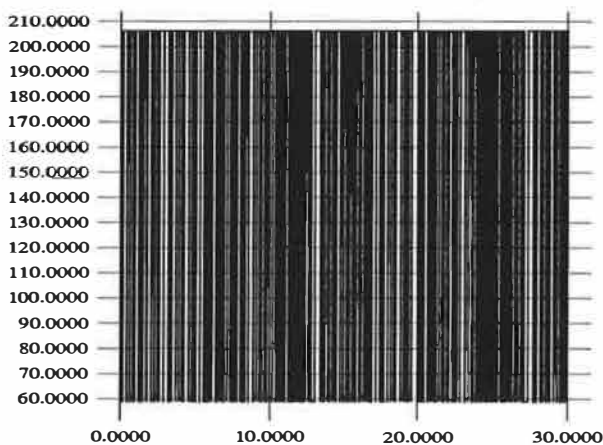


Figure 8: Clock frequency for the MPEG application using the best scheduling policy from our empirical study – the scheduling policy only select 59Mhz or 206MHz clock settings and changes clock settings frequently. This scheduling policy results in suboptimal energy savings but avoids noticeable application slowdown.

The PAST policy we described results in a small but statistically significant reduction in energy for the MPEG application. Allowing the processor to scale the voltage when the clock speed drops below 162.2MHz results in no statistical decrease.

We initially surmised that there is no improvement because the cost of voltage and clock scaling on our platform out-weighs any gains. We measured the cost of clock and voltage scaling using the DAQ. To measure clock scaling, we coded a tight loop that switched the processor clock as quickly as possible.

Before each clock change, we inverted the state of a specific GPIO and used the DAQ to measure the interval with high precision. We took measurements when the clock changed across many different clock settings (*e.g.* from 59 to 206MHz, from 191 to 206MHz and so on).

Clock scaling took approximately 200microseconds, independent of the starting or target speed. During that time, the processor can not execute instructions. Thus, frequency changing varies between 11,200 clock periods at 59MHz and 40,000 clock periods at 200MHz.

We measured the time for the voltage to settle following a voltage change. It takes ≈ 250 microseconds to reduce voltage from 1.5V to 1.23V; in fact, the voltage slowly reduces, drops below 1.23V and then rapidly settles on 1.23V. Voltage increases were effectively instantaneous. We suspect the slow decay occurs because of capacitance; many processors use external decoupling capacitors to provide sufficient current sourcing for processors that have widely varying current demands.

These measurements indicate that the time needed for clock and voltage changes are less than 2% of the scheduling interval; thus, we would be able to change the clock or voltage on every scheduling decision with less than 2% overhead. The fact that we see little energy reduction is related to the limited energy savings possible with the voltage scaling available on this platform and the efficacy of the policies we explored.

6 Conclusions and Future Work

Our implementation results were disappointing to us – we had hoped to be able to identify a prediction heuristic that resulted in significant energy savings, and we thought that the claims made by previous studies would be born out by experimentation. Although we have

Algorithm	Energy
Constant Speed @ 206.4 MHz, 1.5 Volts	85.59 - 86.49
Constant Speed @ 132.7 MHz, 1.5 Volts	79.59 - 80.94
Constant Speed @ 132.7 MHz, 1.23 Volts	73.76 - 74.41
PAST, Peg - Peg, Thresholds: > 98% scales up, < 93% scales down, 1.5 Volts	85.03 - 85.47
PAST, Peg - Peg, Thresholds: > 98% scales up, < 93% scales down, Voltage Scaling @ 162.2 MHz	84.60 - 85.45

Table 2: Summary of Performance of Best Clock Scaling Algorithms

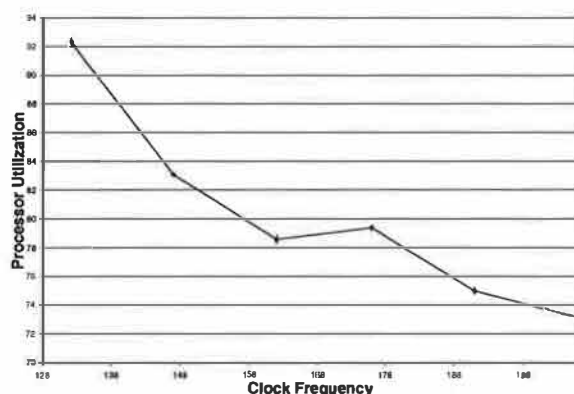


Figure 9: Non-linear change in Utilization with Clock Frequency (in MHz)

Processor Freq.	Cycles/Mem. Reference	Cycles / Cache Reference
59.0	11	39
73.7	11	39
88.5	11	39
103.2	11	39
118.0	13	41
132.7	14	42
147.5	14	49
162.2	15	50
176.9	18	60
191.7	19	61
206.4	20	69

Table 3: Memory access time in cycles for reading individual words as well as full cache lines.

found a policy that saves some energy, that policy leaves much to be desired. The policy causes many voltage and clock changes, which may incur unnecessary overhead; this will be less of a problem as processors are better designed to accommodate those changes. However, the policy did result in both the most responsive system behavior and most significant energy reduction of all the policies we examined.

As with all empirical studies, there are anomalies in our system that we can not explain and that may have influenced our results. We found that the processor utilization does not always vary linearly with clock frequency. Figure 9 shows the processor utilization vs. clock frequency for the MPEG benchmark. There is a distinct “plateau” between 162MHz and 176.9MHz. We believe that this delay may be induced by the varying number of clock cycles needed for memory accesses as the processor frequency changes, as shown in Table 3. That table shows the memory access time for EDODRAM for reading individual words or a full cache line; there is an obvious non-linear increase between 162MHz and 176.9MHz. The potential speed mismatch between processor and memory has been noted by others [12], but we have not devised a way to verify that this is the only factor causing the non-linear behavior we noted.

This paper is the first step on an effort to provide robust support for voltage and clock scheduling within the Linux operating system. Although our initial results are disappointing, we feel that they serve to stop us from attempting to devise clever heuristics that could be used for clock scheduling. It may well be that Pering [11] reached a similar conclusion since their later publications discontinued the use of heuristics, but their publications don’t describe the implementation of their operating system design or the rationale behind the policies used. Furthermore, they don’t describe how deadlines are to be “synthesized” for applications such as Web, TalkingEditor and Web where there is no clear “deadline”.

Our immediate future work is to provide “deadline” mechanisms in Linux. These deadlines are not precisely the same mechanism needed in a true real-time O/S – in a RTOS, the application does not care if the deadline is reached early, while energy scheduling would prefer for the deadline to be met as late as possible. A further challenge we face will be to find a way to automatically synthesize those deadlines for complex applications.

7 Acknowledgments

We would like to thank the members of Compaq’s Palo Alto Research Labs who created the Itsy Pocket Computer and built the infrastructure that made this work possible. We would also like to thank Wayne Mack for adapting the Itsy to fit our needs.

This work was supported in part by NSF grant CCR-9988548, a DARPA contract, and an equipment donation from Compaq Computer.

References

- [1] Jay Heeb. The next generation of strongarm. In *Embedded Processor Forum*. MDR, May 1999.
- [2] Intel Corporation. Moblie pentium iii processor in bga2 and micro-pga2 packages. Datasheet Order #245302-002, 2000.
- [3] David Linden (editor). *Handbook of Batteries*, 2nd ed. McGraw-Hill, New York, 1995.
- [4] Carla-Fabiana Chiasserini and Ramesh R. Rao. Pulsed Battery Discharge in Communication Devices. In *The Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 88–95, August 1999.
- [5] Trevor Pering, Tom Burd, and Robert Brodersen. The simulation of dynamic voltage scaling algorithms. In *IEEE Symposium on Low Power Electronics*. IEEE Symposium on Low Power Electronics, 1995.
- [6] Kinshuk Govil, Edwin Chan, , and Hal Wasserman. Comparing algorithms for dynamic speed-setting of a low-power cpu. In *Proceedings of The First ACM International Conference on Mobile Computing and Networking*, Berkeley, CA, November 1995.
- [7] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced cpu energy. In *First Symposium on Operating Systems Design and Implementation*, pages 13–23, November 1994.
- [8] Trevor Pering. Private communication.
- [9] J. Montanaro and *et. al.* A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor. In *Digital Technical Journal*, volume 9. Digital Equipment Corporation, 1997.
- [10] Dan Dobberpuhl. The Design of a High Performance Low Power Microprocessor. In *International Symposium on Low Power Electronics and Design*, pages 11–16, August 1996.
- [11] Trevor Pering, Tom Burd, and Robert Brodersen. Voltage scheduling in the lparm microprocessor system. In *Proceedings of the 2000 International Symposium on Low Power Design*, August 2000.
- [12] Thomas L. Martin. *Balancing Batteries, Power, and Performance: System Issues in CPU Speed-Setting for Mobile Computers*. PhD thesis, Carnegie Mellon University, 1999.
- [13] Keith I. Farkas, Jason Flinn, Godmar Back, Dirk Grunwald, and Jennifer Anderson. Quantifying the energy consumption of a pocket computer and a java virtual machine. In *Proceedings of the ACM SIGMETRICS ’00 International Conference on Measurement and Modeling of Computer Systems*, 2000. (to appear).
- [14] Transvirtual Technologies Inc. Kaffe Java Virtual Machine. <http://www.transvirtual.com>.

Towards Higher Disk Head Utilization: Extracting Free Bandwidth From Busy Disk Drives

Christopher R. Lumb, Jiri Schindler,
Gregory R. Ganger, David F. Nagle
Carnegie Mellon University
Erik Riedel
Hewlett-Packard Labs

Abstract

Freeblock scheduling is a new approach to utilizing more of a disk's potential media bandwidth. By filling rotational latency periods with useful media transfers, 20–50% of a never-idle disk's bandwidth can often be provided to background applications with no effect on foreground response times. This paper describes freeblock scheduling and demonstrates its value with simulation studies of two concrete applications: segment cleaning and data mining. Free segment cleaning often allows an LFS file system to maintain its ideal write performance when cleaning overheads would otherwise reduce performance by up to a factor of three. Free data mining can achieve over 47 full disk scans per day on an active transaction processing system, with no effect on its disk performance.

1 Introduction

Disk drives increasingly limit performance in many computer systems, creating complexity and restricting functionality. However, in recent years, the rate of improvement in media bandwidth (40+% per year) has stayed close to that of computer system attributes that are driven by Moore's Law. It is only the mechanical positioning aspects (i.e., seek times and rotation speeds) that fail to keep pace. If 100% utilization of the potential media bandwidth could be realized, disk performance would scale roughly in proportion to the rest of the system over time. Unfortunately, utilizations of 2–15% are more commonly observed in practice.

This paper describes and analyzes a new approach, called *freeblock scheduling*, to increasing media bandwidth utilization. By interleaving low priority disk activity with the normal workload (here referred to as background and foreground, respectively), one can replace many foreground rotational latency de-

lays with useful background media transfers. With appropriate freeblock scheduling, background tasks can receive 20–50% of a disk's potential media bandwidth without any increase in foreground request service times. Thus, this background disk activity is completed for free during the mechanical positioning for foreground requests.

There are many disk-intensive background tasks that are designed to occur during otherwise idle time. Examples include disk reorganization, file system cleaning, backup, prefetching, write-back, integrity checking, RAID scrubbing, virus detection, tamper detection, report generation, and index reorganization. When idle time does not present itself, these tasks either compete with foreground tasks or are simply not completed. Further, when they do compete with other tasks, these background tasks do not take full advantage of their relatively loose time constraints and paucity of sequencing requirements. As a result, these "idle time" tasks often cause performance or functionality problems in busy systems. With freeblock scheduling, background tasks can operate continuously and efficiently, even when they do not have the system to themselves.

This paper quantifies the effects of disk, workload, and disk scheduling algorithms on potential free bandwidth. Algorithms are developed for increasing the available free bandwidth and for efficient freeblock scheduling. For example, with less than a 6% increase in average foreground access time, a Shortest-Positioning-Time-First scheduling algorithm that favors reduction of seek time over reduction of rotational latency can provide an additional 66% of free bandwidth. Experiments also show that freeblock scheduling decisions can be made efficiently enough to be effective in highly loaded systems.

This paper uses simulation to explore freeblock scheduling, demonstrating its value with concrete

examples of its use for storage system management and disk-intensive applications. The first example shows that cleaning in a log-structured file system can be done for free even when there is no truly idle time, resulting in up to a 300% speedup. The second example explores the use of free bandwidth for data mining on an active on-line transaction processing (OLTP) system, showing that over 47 full scans per day of a 9GB disk can be made with no impact on OLTP performance.

In a recent paper [45], we proposed a scheme for performing data mining “for free” on a busy OLTP system. The scheme combines Active Disks [46] with use of idle time and aggressive interleaving of data mining requests with OLTP requests. This paper generalizes and extends the latter, developing an understanding of free bandwidth availability and exploring its use.

The remainder of this paper is organized as follows. Section 2 describes free bandwidth and discusses its use in systems. Section 3 quantifies the availability of potential free bandwidth and how it varies with disk characteristics, foreground workloads, and foreground disk scheduling algorithms. Section 4 describes our freeblock scheduling algorithm. Section 5 evaluates the use of free bandwidth for cleaning of LFS log segments. Section 6 evaluates the use of free bandwidth for data mining of active OLTP systems. Section 7 discusses related work. Section 8 summarizes the paper’s contributions.

2 Free Bandwidth

At a high-level, the time required for a disk media access, T_{access} , can be computed as

$$T_{access} = T_{seek} + T_{rotate} + T_{transfer}$$

Of T_{access} , only the $T_{transfer}$ component represents useful utilization of the disk head. Unfortunately, the other two components generally dominate. Many data placement and scheduling algorithms have been devised to increase disk head utilization by increasing transfer sizes and reducing positioning overheads. Freeblock scheduling complements these techniques by transferring additional data during the T_{rotate} component of T_{access} .

Fundamentally, the only time the disk head cannot be transferring data sectors to or from the media is during a seek. In fact, in most modern disk drives, the firmware will transfer a large request’s data to or from the media “out of order” to minimize wasted time; this feature is sometimes referred to as zero-latency or immediate access. While seeks

are unavoidable costs associated with accessing desired data locations, rotational latency is an artifact of not doing something more useful with the disk head. Since disk platters rotate constantly, a given sector will rotate past the disk head at a given time, independent of what the disk head is doing up until that time. So, there is an opportunity to do something more useful than just waiting for desired sectors to arrive at the disk head.

Freeblock scheduling consists of predicting how much rotational latency will occur before the next foreground media transfer, squeezing some additional media transfers into that time, and still getting to the destination track in time for the foreground transfer. The additional media transfers may be on the current or destination tracks, on another track near the two, or anywhere between them, as illustrated in Figure 1. In the two latter cases, additional seek overheads are incurred, reducing the actual time available for the additional media transfers, but not completely eliminating it.

Accurately predicting future rotational latencies requires detailed knowledge of many disk performance attributes, including layout algorithms and time-dependent mechanical positioning overheads. These predictions can utilize the same basic algorithms and information that most modern disks employ for their internal scheduling decisions, which are based on overall positioning overheads (seek time plus rotational latency) [50, 30]. However, this may require that freeblock scheduling decisions be made by disk firmware. Fortunately, the increasing processing capabilities of disk drives [1, 22, 32, 46] make advanced on-drive storage management feasible [22, 57].

2.1 Using Free Bandwidth

Potential free bandwidth exists in the time gaps that would otherwise be rotational latency delays for foreground requests. Therefore, freeblock scheduling must opportunistically match these potential free bandwidth sources to real bandwidth needs that can be met within the given time gaps. The tasks that will utilize the largest fraction of potential free bandwidth are those that provide the freeblock scheduler with the most flexibility. Tasks that best fit the freeblock scheduling model have low priority, large sets of desired blocks, no particular order of access, and small working memory footprints.

Low priority. Free bandwidth is inherently in the background, and freeblock requests will only be serviced when opportunities arise. Therefore, response times may be extremely long for such requests, mak-

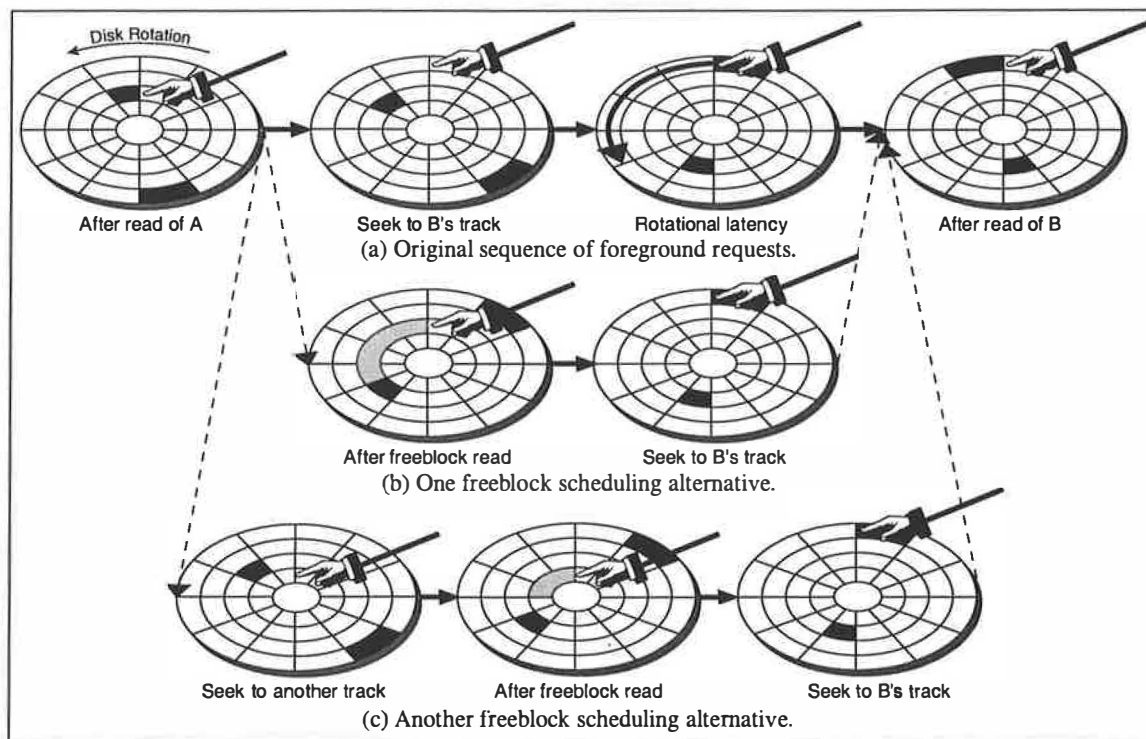


Figure 1: Illustration of two freeblock scheduling possibilities. Three sequences of steps are shown, each starting after completing the foreground request to block *A* and finishing after completing the foreground request to block *B*. Each step shows the position of the disk platter, the read/write head (shown by the pointer), and the two foreground requests (in black) after a partial rotation. The top row, labelled (a), shows the default sequence of disk head actions for servicing request *B*, which includes 4 sectors worth of potential free bandwidth (a.k.a. rotational latency). The second row, labelled (b), shows free reading of 4 blocks on *A*'s track using 100% of the potential free bandwidth. The third row, labelled (c), shows free reading of 3 blocks on another track, yielding 75% of the potential free bandwidth.

ing them most appropriate for background activities. Further, freeblock scheduling is not appropriate for a set of equally important requests; splitting such a set between a foreground queue and a freeblock queue reduces the options of both schedulers. All such requests should be considered by a single scheduler.

Large sets of desired blocks. Since freeblock schedulers work with restricted free bandwidth opportunities, their effectiveness tends to increase when they have more options. That is, the larger the set of disk locations that are desired, the higher the probability that a free bandwidth opportunity can be matched to a need. Therefore, tasks that involve larger fractions of the disk's capacity generally utilize larger fractions of the potential free bandwidth.

No particular order of access. Ordering requirements restrict the set of requests that can be considered by the scheduler at any point in time. Since the effectiveness of freeblock scheduling is directly related to the number of outstanding requests, workloads with little or no ordering requirements tend to

utilize more of the potential free bandwidth.

Small working memory footprints. Significant need to buffer multiple blocks before processing them creates artificial ordering requirements due to memory limitations. Workloads that can immediately process and discard data from freeblock requests tend to be able to request more of their needed data at once.

To clarify the types of tasks that fit the freeblock scheduling model, Table 1 presents a sample interface for a freeblock scheduling subsystem, ignoring component and protection boundary issues. This interface is meant to be illustrative only; a comprehensive API would need to address memory allocation, protection, and other issues.

This sample freeblock API has four important characteristics. First, no call into the freeblock scheduling subsystem waits for a disk access. Instead, calls to register requests return immediately, and subsequent callbacks report request completions. This allows applications to register large sets of freeblock

Function Name	Arguments	Description
<i>freeblock_readblocks</i>	<i>diskaddrs, blksize, callback</i>	Register freeblock read request(s)
<i>freeblock_writeblocks</i>	<i>diskaddrs, blksize, buffers, callback</i>	Register freeblock write request(s)
<i>freeblock_abort</i>	<i>diskaddrs, blksize</i>	Abort registered freeblock request(s)
<i>freeblock_promote</i>	<i>diskaddrs, blksize</i>	Promote registered freeblock request(s)
<i>*(callback)</i>	<i>diskaddr, blksize, buffer</i>	Call back to task with desired block

Table 1: A simple interface to a freeblock subsystem. *freeblock_readblocks* and *freeblock_writeblocks* register one or more single-block freeblock requests, with an application-defined block size. *freeblock_abort* and *freeblock_promote* are applied to previously registered requests, to either cancel pending freeblock requests or convert them to foreground requests. When promoted, multiple contiguous freeblock requests can be merged into a single foreground request. **(callback)* is called by the freeblock subsystem to report availability (or write completion) of a single previously-requested block. When the request was a read, *buffer* points to a buffer containing the desired data. The freeblock subsystem reclaims this buffer when **(callback)* returns, meaning that the callee must either process the data immediately or copy it to another location before returning control.

requests. Second, block sizes are provided with each freeblock request, allowing applications to ensure that useful units are provided to them. Third, freeblock read requests do not specify memory locations for read data. Completion callbacks provide pointers to buffers owned by the freeblock scheduling subsystem and indicate which requested data blocks are in them. This allows tasks to register many more freeblock reads than their memory resources would otherwise allow, giving greater flexibility to the freeblock scheduling subsystem. For example, the data mining example in Section 6 starts by registering freeblock reads for all blocks on the disk. Fourth, freeblock requests can be aborted or promoted to foreground requests at any time. The former allows tasks to register for more data than are absolutely required (e.g., a search that only needs one match). The latter allows tasks to increase the priority of freeblock requests that may soon impact foreground task performance (e.g., a space compression task that has not made sufficient progress).

2.2 Applications

Freeblock scheduling is a new tool, and we expect that system designers will find many unanticipated uses for it. This section describes some of the applications we see for its use.

Scanning applications. In many systems, there are a variety of support tasks that scan large portions of disk contents. Such activities are of direct benefit to users, although they may not be the highest priority of the system. Examples of such tasks include report generation, RAID scrubbing, virus detection, tamper detection [33], and backup. Section 6 explores data mining of an active transaction processing system as a concrete example of such use of free bandwidth.

These disk-scanning application tasks are ideal candidates for free bandwidth utilization. Appropriately structured, they can exhibit all four of the desirable characteristics discussed above. For example, report generation tasks (and data mining in general) often consist of collecting statistics about large sets of small, independent records. These tasks may be of lower priority than foreground transactions, access a large set of blocks, involve no ordering requirements, and process records immediately. Similarly, virus detectors examine large sets of files for known patterns. The files can be examined in any order, though internal statistics for partially-checked files may have significant memory requirements when pieces of files are read in no particular order. Backup applications can be based on physical format, allowing flexible block ordering with appropriate indices, though single-file restoration is often less efficient [28, 14]. Least flexible of these examples would be tamper detection that compares current versions of data to “safe” versions. While the comparisons can be performed in any order, both versions of a particular datum must be available in memory to complete a comparison. Memory limitations are unlikely to allow arbitrary flexibility in this case.

Internal storage optimization. Another promising use for free bandwidth is internal storage system optimization. Many techniques have been developed for reorganizing stored data to improve performance of future accesses. Examples include placing related data contiguously for sequential disk access [37, 57], placing hot data near the center of the disk [56, 48, 3], and replicating data on disk to provide quicker-to-access options for subsequent reads [42, 61]. Other examples include index reorganization [29, 23] and compression of cold data [11].

Section 5 explores segment cleaning in log-structured file systems as a concrete example of such use of free bandwidth.

Although internal storage optimization activities exhibit the first two qualities listed in Section 2.1, they can impose some ordering and memory restrictions on media accesses. For example, reorganization generally requires clearing (i.e., reading or moving) destination regions before different data can be written. Also, after opportunistically reading data for reorganization, the task must write this data to their new locations. Eventually, progress will be limited by the rate at which these writes can be completed, since available memory resources for buffering such data are finite.

Prefetching and Prewriting. Another use of free bandwidth is for anticipatory disk activities such as prefetching and prewriting. Prefetching is well-understood to offer significant performance enhancements [44, 9, 25, 36, 54]. Free bandwidth prefetching should increase performance further by avoiding interference with foreground requests and by minimizing the opportunity cost of aggressive predictions. As one example, the sequence shown in Figure 1(b) shows one way that the prefetching common in disk firmware could be extended with free bandwidth. Still, the amount of prefetched data is necessarily limited by the amount of memory available for caching, restricting the number of freeblock requests that can be issued.

Prewriting is the same concept in reverse. That is, prewriting is early writing out of dirty blocks under the assumption that they will not be overwritten or deleted before write-back is actually necessary. As with prefetching, the value of prewriting and its relationship with non-volatile memory are well-known [4, 10, 6, 23]. Free bandwidth prewriting has the same basic benefits and limitations as free prefetching.

3 Availability of Free Bandwidth

This section quantifies the availability of potential free bandwidth, which is equal to a disk's total potential bandwidth multiplied by the fraction of time it spends on rotational latency delays. The amount of rotational latency depends on a number of disk, workload, and scheduling algorithm characteristics.

The experimental data in this section was generated with the DiskSim simulator [20], which has

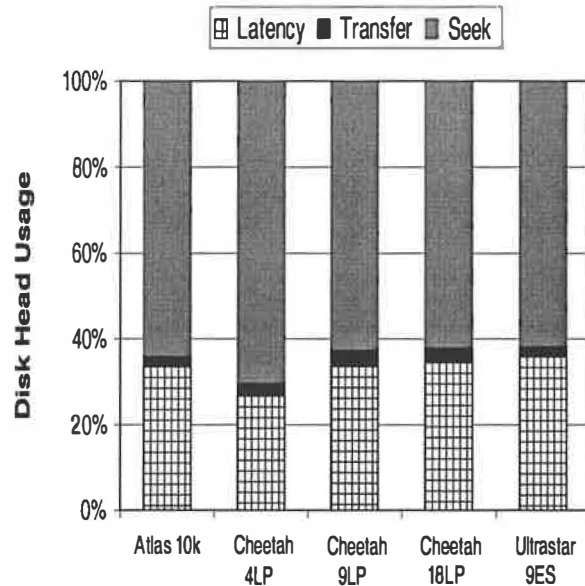


Figure 2: **Disk head usage for several modern disks.** The cross-hatch portion, representing rotational latency, indicates the percentage of total disk bandwidth available as potential free bandwidth.

been shown to accurately model several modern disk drives [17], including those explored here. By default, the experiments use a Quantum Atlas 10K disk drive and a synthetic workload referred to as *random*. This random workload consists of 10,000 foreground requests issued one at a time with no idle time between requests (closed system arrival model with no think time). Other default parameters for the random workload are request size of 4KB, uniform distribution of starting locations across the disk capacity, and 2:1 ratio of reads to writes.

Most of the bar graphs presented in this section have a common structure. Each bar breaks down disk head usage into several regions that add up to 100%, with each region representing the percentage of the total attributed to the corresponding activity. All such bars include regions for foreground seek times, rotational latencies, and media transfers. The rotational latency region represents the potential free bandwidth (as a percentage of the disk's total bandwidth) available for the disk-workload combination.

3.1 Impact of disk characteristics

Figure 2 shows breakdowns of disk head usage for five modern disk drives whose basic characteristics are given in Table 2. Overall, for the random workload, about one third (27–36%) of each disk's head usage can be attributed to rotational latency. Thus,

	Quantum Atlas 10K	Seagate Cheetah 4LP	Seagate Cheetah 9LP	Seagate Cheetah 18LP	IBM Ultrastar 9ES
Year	1999	1996	1997	1998	1998
Capacity	9 GB	4.5 GB	9 GB	9 GB	9 GB
Cylinders	10042	6581	6962	9772	11474
Tracks per cylinder	6	8	12	6	5
Sectors per track	229-334	131-195	167-254	252-360	247-390
Spindle speed (RPM)	10025	10033	10025	10025	7200
Average seek	5.0 ms	7.7 ms	5.4 ms	5.2 ms	7.0 ms
Min-Max seeks	1.2-10.8 ms	0.6-16.1 ms	0.8-10.6 ms	0.7-10.8 ms	1.1-12.7 ms

Table 2: Basic characteristics of several modern disk drives.

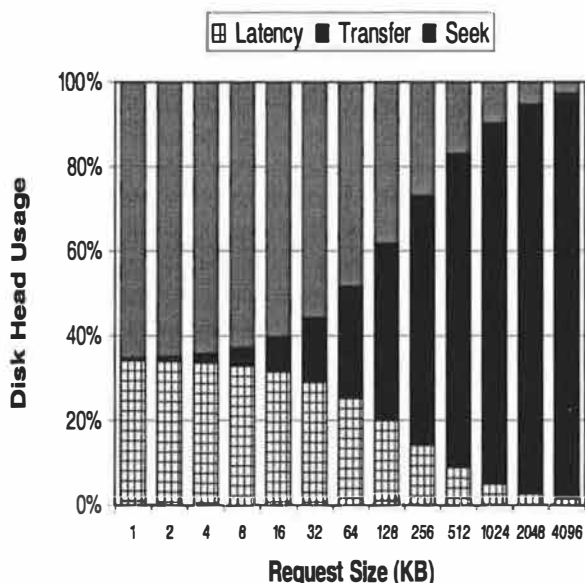


Figure 3: Disk head usage as a function of request size.

about one third of the media bandwidth is available for freeblock scheduling, even with no inter-request locality. At a more detailed level, the effect of key disk characteristics can be seen in the breakdowns. For example, the faster seeks of the Cheetah 9LP, relative to the Cheetah 4LP, can be seen in the smaller seek component.

3.2 Impact of workload characteristics

Figure 3 shows how the breakdown of disk head usage changes as the request size of the random workload increases. As expected, larger request sizes yield larger media transfer components, reducing the seek and latency components by amortizing larger transfers over each positioning step. Still, even for large random requests (e.g., 256KB), disk head uti-

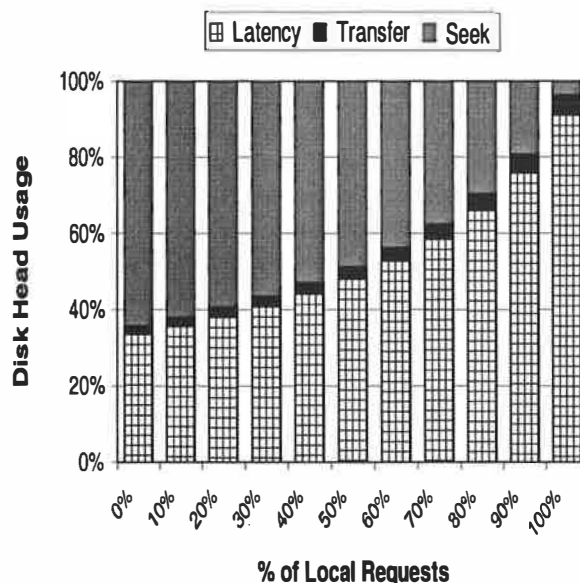


Figure 4: Disk head usage as a function of access locality. The default workload was modified such that a percentage of request starting locations are "local" (taken from a normal distribution centered on the last requested location, with a standard deviation of 4MB). The remaining requests are uniformly distributed across the disk's capacity. This locality model crudely approximates the effect of "cylinder group" layouts [38] on file system workloads.

lization is less than 55% and potential free bandwidth is 15%.

Figure 4 shows how the breakdown of disk head usage changes as the degree of access locality increases. Because access locality tends to reduce seek distances without directly affecting the other components, this graph shows that both the transfer and latency components increase. For example, when 70% of the requests are within the same "cylinder group" [38] as the last request, almost 60% of the disk head's time is spent in rotational latency and

is thus available as free bandwidth. Since disk access locality is a common attribute of many environments, one can generally expect more potential free bandwidth than the 33% predicted for the *random* workload.

Figure 4 does not show the downside (for freeblock scheduling) of high degrees of locality — starvation of distant freeblock requests. That is, if foreground requests keep the disk head in one part of the disk, it becomes difficult for a freeblock scheduler to successfully make progress on freeblock requests in distant parts of the disk. This effect is taken into account in the experiments of Sections 5 and 6.

3.3 Impact of scheduling algorithm

Figure 5 shows how the breakdown of disk head usage changes for different scheduling algorithms applied to foreground requests. Specifically, four scheduling algorithms are shown: First-Come-First-Served (FCFS), Circular-LOOK (C-LOOK), Shortest-Seek-Time-First (SSTF), and Shortest-Positioning-Time-First (SPTF). FCFS serves requests in arrival order. C-LOOK [49] selects the next request in ascending starting address order; if none exists, it selects the request with the lowest starting address. SSTF [16] selects the request that will incur the shortest seek. SPTF [30, 50, 60] selects the request that will incur the smallest overall positioning delay (seek time plus rotational latency).

On average, C-LOOK and SSTF reduce seek times without affecting transfer times and rotational latencies. Therefore, we expect (and observe) the seek component to decrease and the other two to increase. In fact, for this workload, the rotational latency component increases to 50% of the disk head usage. On the other hand, SPTF tends to decrease both overhead components, and Figure 5 shows that the rotational latency component decreases significantly (to 22%) relative to the other scheduling algorithms.

SPTF requires the same basic time predictions as freeblock scheduling. Therefore, its superior performance will make it a common foreground scheduling algorithm in systems that can support freeblock scheduling, making its effect on potential free bandwidth a concern. To counter this effect, we propose a modified SPTF algorithm that is weighted to select requests with both small total positioning delays and large rotational latency components. The algorithm, here referred to as SPTF-SW $n\%$, selects the request with the smallest seek time component among the pending requests whose positioning times are within $n\%$ of the shortest positioning time. So, logically,

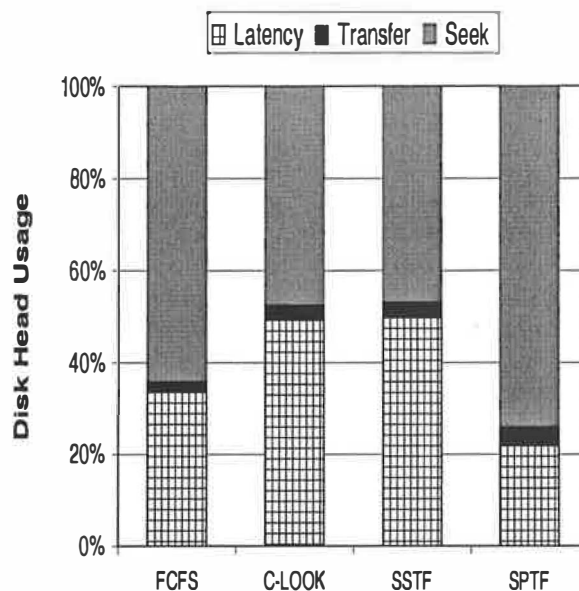


Figure 5: Disk head usage for several foreground scheduling algorithms. The default workload was modified to always have 20 requests outstanding. Lowering the number of outstanding requests reduces the differences between the scheduling algorithms, as they all converge on FCFS.

this algorithm first uses the standard SPTF algorithm to identify the next most efficient request, denoted A , to be scheduled. Then, it makes a second pass to find the pending request, denoted B , that has the smallest seek time while still having a total positioning time within $n\%$ of A 's. Request B is then selected and scheduled. The actual implementation makes a single pass, and its measured computational overhead is only 2–5% higher than that of SPTF. This algorithm creates a continuum between SPTF (when $n = 0$) and SSTF (when $n = \infty$), and we expect the disk head usage breakdown to reflect this.

Figure 6 shows the breakdown of disk head usage and the average foreground request access time when SPTF-SW $n\%$ is used for foreground request scheduling. As expected, different values of n result in a range of options between SPTF and SSTF. As n increases, seek reduction becomes a priority, and the rotational latency component of disk head usage increases. At the same time, average access times increase as total positioning time plays a less dominant role in the decision process. Fortunately, the benefits increase rapidly before experiencing diminishing returns, and the penalties increase slowly before ramping up. So, using SPTF-SW40% as an example, we see that a 6% increase in average access

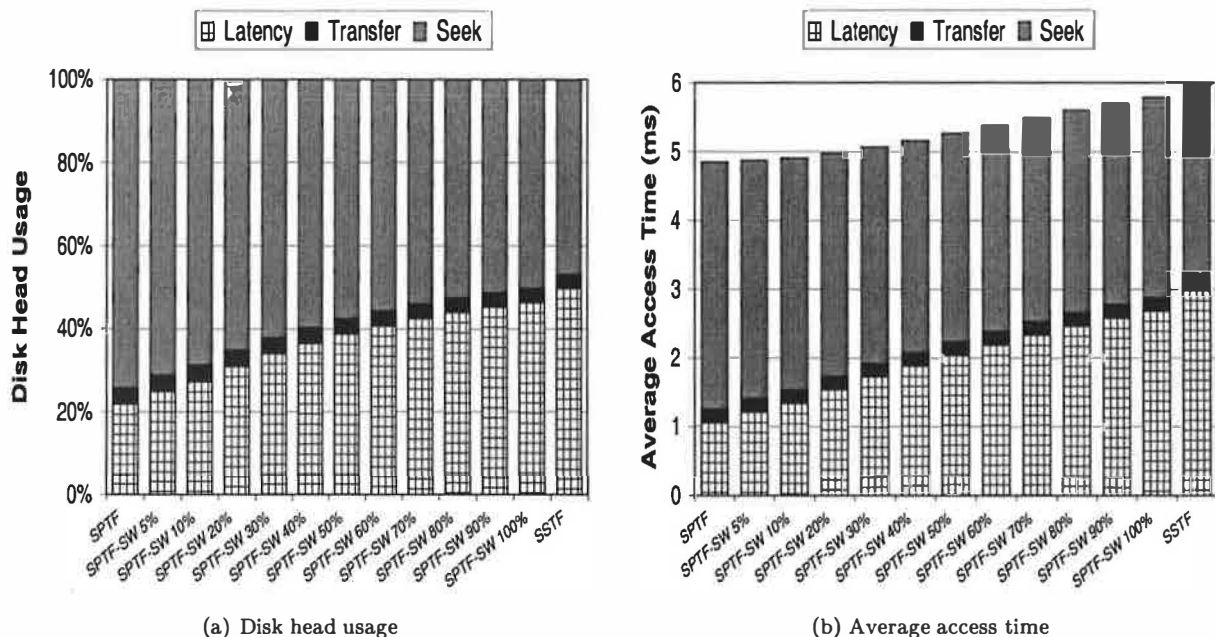


Figure 6: Disk head usage and average access time with SPTF-SW $n\%$ for foreground scheduling. The default workload was modified to always have 20 requests outstanding.

time can provide 66% more potential free bandwidth (i.e., 36% rotational latency for SPTF-SW40% compared to SPTF's 22%). This represents half of the free bandwidth difference between SPTF and SSTF at much less than the 25% foreground access time difference.

4 Freeblock Scheduling Decisions

Freeblock scheduling is the process of identifying free bandwidth opportunities and matching them to pending freeblock requests. This section describes and evaluates the computational overhead of the freeblock scheduling algorithm used in our experiments.

Our freeblock scheduler works independently of the foreground scheduler and maintains separate data structures. After the foreground scheduler chooses the next request, B , the freeblock scheduler is invoked. It begins by computing the rotational latency that would be incurred in servicing B ; this is the free bandwidth opportunity. This computation requires accurate estimates of disk geometry, current head position, seek times, and rotation speed. The freeblock scheduler then searches its list of pending freeblock requests for the most complete use of this opportunity; that is, our freeblock scheduler greedily schedules freeblock requests within free bandwidth

opportunities based on the number of blocks that can be accessed.

Our current freeblock scheduler assumes that the most complete use of a free bandwidth opportunity is the maximal answer to the question, "for each track on the disk, how many desired blocks could be accessed in this opportunity?". For each track, t , answering this question requires computing the extra seek time involved with seeking to t and then seeking to B 's track, as compared to seeking directly to B 's track. Answering this question also requires determining which disk blocks will pass under the head during the remaining rotational latency time and counting how many of them correspond to pending freeblock requests. Note that no extra seek is required for the source track or for B 's track.

Obviously, such an exhaustive search can be extremely time consuming. We prune the search space in several ways. First, the freeblock scheduler skips all tracks for which the number of desired blocks is less than the best value found so far. Second, the freeblock scheduler only considers tracks for which the remaining free bandwidth (after extra seek overheads) is greater than the best value found so far. Third, the freeblock scheduler starts by searching the source and destination cylinders (from the previous and current foreground requests), which yield the best choices whenever they are fully populated,

and then searching in ascending order of extra seek time. Combined with the first two pruning steps, this ordered search frequently terminates quickly.

The algorithm described above performs well when there is a large number of pending freeblock requests. For example, when 20–100% of the disk is desired, freeblock scheduling decisions are made in 0–2.5ms on a 550MHz Intel Pentium III, which is much less than average disk access times. For such cases, it should be possible to schedule the next freeblock request in real-time before the current foreground request completes, even with a less-powerful CPU. With greater fragmentation of freeblock requests, the time required for the freeblock scheduler to make a decision rises significantly. The worst-case computation time of this algorithm occurs when there are large numbers of small requests evenly distributed across all cylinders. In this case, the algorithm searches a large percentage of the available disk space in the hopes of finding a larger section of blocks than it has already found. To address this problem, one can simply halt searches after some amount of time (e.g., the time available before the previous foreground request completes). In most cases, this has a negligible effect on the achieved free bandwidth. For all experiments in this paper, the freeblock scheduling algorithm was only allowed to search for the next freeblock request in the time that the current foreground request was being serviced.

The base algorithm described here enables significant use of free bandwidth, as shown in subsequent sections. Nonetheless, development of more efficient and more effective freeblock scheduling algorithms is an important area for further work. This will include using both free bandwidth and idle time for background tasks; the algorithm above and all experiments in this paper use only free bandwidth.

5 Free cleaning of LFS segments

The log-structured file system [47] (LFS) was designed to reduce the cost of disk writes. Towards this end, it remaps all new versions of data into large, contiguous regions called segments. Each segment is written to disk with a single I/O operation, amortizing the cost of a single seek and rotational delay over a write of a large number of blocks. A significant challenge for LFS is ensuring that empty segments are always available for new data. LFS answers this challenge with an internal defragmentation operation called *cleaning*. Ideally, all necessary cleaning would be completed during idle time, but this is not

always possible in a busy system. The potential and actual penalties associated with cleaning have been the subject of heated debate [52] and several research efforts [51, 37, 7, 39, 59]. With freeblock scheduling, the cost of segment cleaning can be close to zero for many workloads.

5.1 Design

Cleaning of a previously written segment involves identifying the subset of live blocks, reading them into memory, and writing them into the next segment. Live blocks are those that have not been overwritten or deleted by later operations; they can be identified by examining the on-disk segment summary structure to determine the original identity of each block (e.g., block 4 of file 3) and then examining the auxiliary structure for the block's original owner (e.g., file 3's i-node). Segment summaries, auxiliary structures, and live blocks can be read via freeblock requests. There are ordering requirements among these, but live blocks can be read in any order and moved into their new locations immediately.

Like other background LFS cleaners, our freeblock segment cleaner is invoked when the number of empty segments drops below a certain threshold. When invoked, the freeblock cleaner selects several non-empty segments and uses freeblock requests to clean them in parallel with other foreground requests. Cleaning several segments in parallel provides more requests and greater flexibility to the freeblock scheduler. If the freeblock cleaner is not effective enough, the foreground cleaner will be activated when the minimum threshold of free segments is reached.

As live blocks in targeted segments are fetched, they are copied into the in-memory segment that is currently being constructed by LFS writes. Because the live blocks are written into the same segment as data of foreground LFS requests, this method of cleaning is not entirely for free. The auxiliary data structure (e.g., i-node) that marks the location of the block is updated to point to the block's new location in the new segment. When all live blocks are cleaned from a segment on the disk, that segment becomes available for subsequent use.

5.2 Experimental Setup

To experiment with freeblock cleaning, we have modified a log-structured logical disk, called LLD [15]. LLD uses segments consisting of 128 4KB blocks, of which 127 blocks are used for data and one block is used for segment summary. The

default implementation of LLD invokes its cleaner only when the number of free segments drops below a threshold (set to two segments). It does not implement background cleaning. Thus, all segment cleaning activity interferes with the foreground disk I/O. We replaced LLD's default segment selection algorithm for cleaning with Sprite LFS's cost-benefit algorithm[47], yielding better performance for all of the cleaners.

Our experiments were run under Linux 2.2.14 with a combination of real processing times and simulated I/O times provided by DiskSim. To accomplish this, we merged LLD with DiskSim. Computation times between disk I/Os are measured with `gettimeofday`, which uses the Pentium cycle counter. These computation times are used to advance simulation time in DiskSim. DiskSim callbacks report request completions, which are forwarded into the LLD code as interrupts. The contents of the simulated disk are stored in a regular file, and the time required to access this file is excluded from the reported results.

All experiments were run on a 550 MHz Intel Pentium III machine with 256MB of memory. DiskSim was configured to model a modified Quantum Atlas 10K disk. Specifically, since the maximal size of an LLD disk is 400MB, we modified the Atlas 10K specifications to have only one data surface, resulting in a capacity of 1.5GB. Thus, the LLD "partition" occupies about 1/4 of the disk.

To assess the effectiveness of the freeblock cleaner, we used the Postmark v. 1.11 benchmark, which simulates the small-file activity predominant on busy Internet servers [31]. Postmark initially creates a pool of files, then performs a series of transactions, and finally deletes all files created during the benchmark run. A single transaction is one access to an existing file (i.e., read or append) and one file manipulation (i.e., file creation or deletion). We used the following parameter values: 5–10KB file size (default Postmark value), 25000 transactions, and 100 subdirectories. The ratios of read-to-write and create-to-delete were kept at their default values of 1:1. The number of files in the initial pool was varied to provide a range of file system capacity utilizations.

To age the file system, we run the transaction phase twice and report measurements for only the second iteration. The rationale for running the set of transactions the first time is to spread the blocks of the file system among the segments in order to more closely resemble steady-state operation. Recall that Postmark first creates all files before doing transactions

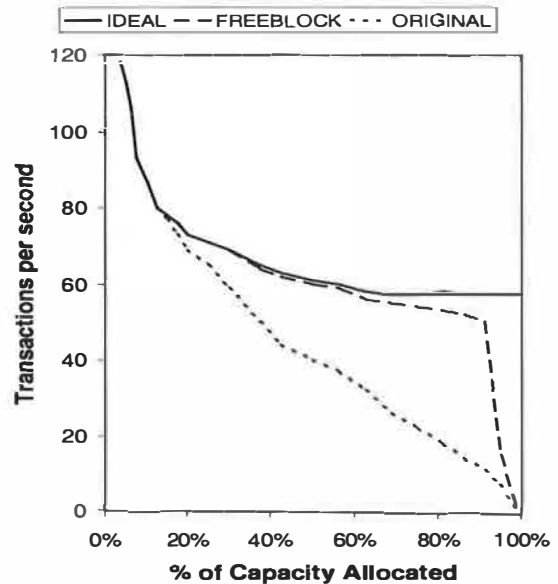


Figure 7: LLD performance for three cleaning strategies. Even with a heavy foreground workload (Postmark), segment cleaning can be completed with just freeblock requests until the file system is 93% full.

which results in all segments being either completely full or completely empty — a situation very unlikely in normal operation.

5.3 Results

Figure 7 shows Postmark's performance for three different cleaner configurations: ORIGINAL is the default LLD cleaner with the Sprite LFS segment selection algorithm. FREEBLOCK is the freeblock cleaner, in which cleaning reads are freeblock requests and cleaning writes are foreground requests. IDEAL subtracts all cleaning costs from ORIGINAL and computes the corresponding throughput, which is unrealistic because infinitely fast foreground cleaning is not possible.

Figure 7 shows the transactions per second for different file system space utilizations, corresponding to different numbers of files initially created by Postmark. The high throughput for low utilizations (less than 8% of capacity) is due to the LLD buffer cache, which absorbs all of the disk activity. IDEAL's performance decreases as capacity utilization increases, because the larger set of files results in fewer cache hits for Postmark's random file accesses. As disk utilization increases, ORIGINAL's throughput decreases consistently due to cleaning overheads, halving performance at 60% capacity and quartering it at 85%. FREEBLOCK maintains performance close to IDEAL (up to 93% utilization). After 93%, there

is insufficient time for freeblock cleaning to keep up with the heavy foreground workload, and the performance of FREEBLOCK degrades as the foreground cleaner increasingly dominates performance. FREEBLOCK's slow divergence from IDEAL between 40% and 93% occurs because FREEBLOCK is being charged for the write cost of cleaned segments while IDEAL is not.

6 Free data mining on OLTP systems

The use of data mining to identify patterns in large databases is becoming increasingly popular over a wide range of application domains and datasets [19, 12, 58]. One of the major obstacles to starting a data mining project within an organization is the high initial cost of purchasing the necessary hardware. Specifically, the most common strategy for data mining on a set of transaction data is to purchase a second database system, copy the transaction records from the OLTP system to the second system each evening, and perform mining tasks only on the second system. This strategy can double capital and operating expenses. It also requires that a company gamble a sizable up-front investment to test suspicions that there may be interesting "nuggets" to be mined from their OLTP databases. With freeblock scheduling, significant mining bandwidth can be extracted from the original system without affecting the original transaction processing activity [45].

6.1 Design

Data mining involves examining large sets of records for statistical features and correlations. Many data mining operations, including nearest neighbor search, association rules [2], ratio and singular value decomposition [34], and clustering [62, 26], eventually translate into a few scans of the entire dataset. Further, individual records can be processed immediately and in any order, matching three of the criteria of appropriate free bandwidth uses.

Our freeblock mining example issues a single freeblock read request for each scan. This freeblock request asks for the entire contents of the database in page-sized chunks. The freeblock scheduler ensures that only blocks of the specified size are provided and that all the blocks requested are read exactly once. However, the order in which the blocks are read will be an artifact of the pattern of foreground OLTP requests.

Interestingly, this same design is appropriate for

some other storage activities. For example, RAID scrubbing consists of verifying that each disk sector can be read successfully (i.e., that no sector has fallen victim to media corruption). Also, a physical backup consists of reading all disk sectors so that they can be written to another device. The free bandwidth achieved for such scanning activities would match that shown for freeblock data mining in this section.

6.2 Experimental Setup

The experiments in Section 6.3 were conducted using the DiskSim simulator configured to model the Quantum Atlas 10K and a synthetic foreground workload based on approximations of observed OLTP workload characteristics. The synthetic workload models a closed system with per-task disk requests separated by think times of 30 milliseconds. We vary the multiprogramming level (MPL), or number of tasks, of the OLTP workload to create increasing foreground load on the system. For example, a multiprogramming level of ten means that there are ten requests active in the system at any given point, either queued at the disk or waiting in think time. The OLTP requests are uniformly distributed across the disk's capacity with a read to write ratio of 2:1 and a request size that is a multiple of 4 kilobytes chosen from an exponential distribution with a mean of 8 kilobytes. Validation experiments (in [45]) show that this workload is sufficiently similar to disk traces of Microsoft's SQL server running TPC-C for the overall freeblock-related insights to apply to more realistic OLTP environments. The background data mining workload uses free bandwidth to make full scans of the disk's contents in 4 KB blocks, completing one scan before starting the next. All simulations run for the time required for the background data mining workload to complete ten full disk scans, and the results presented are averages across these ten scans. The experiments ignore bus bandwidth and record processing overheads, assuming that media scan times dominate; this assumption might be appropriate if the mining data is delivered over distinct buses to dedicated processors either on a small mining system or in Active Disks.

6.3 Results

Figure 8 shows the disk head usage for the foreground OLTP workload at a range of MPLs and the free bandwidth achieved by the data mining task. Low OLTP loads result in low data mining throughput, because little potential free bandwidth exists

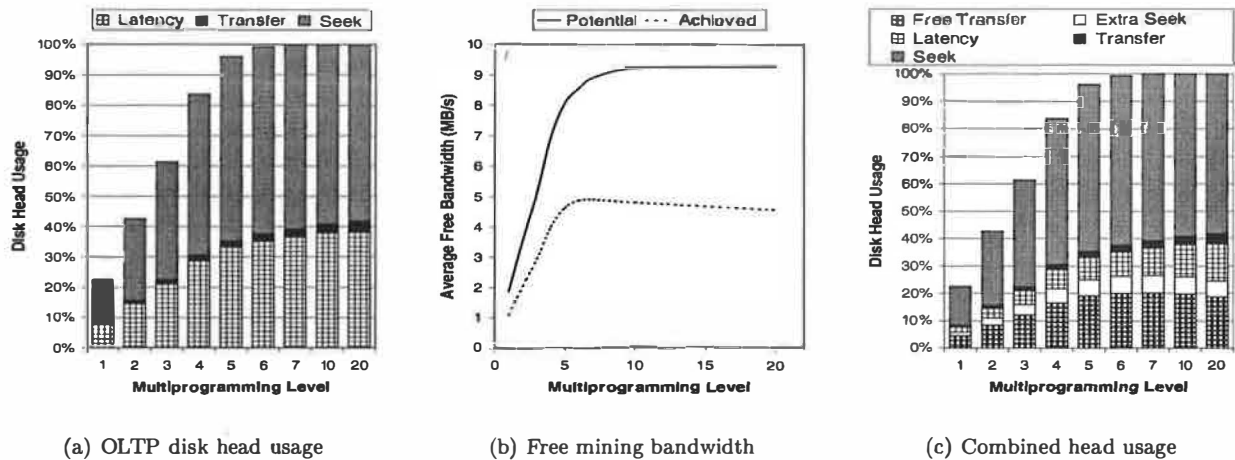


Figure 8: **Average freeblock-based data mining performance.** (a) shows the disk head usage breakdown for the foreground OLTP workload at various MPLs. (b) shows the overall free bandwidth delivered to the data mining application for the same points. (c) shows the disk head usage breakdown with both the foreground OLTP workload and the background data mining application.

when there are few foreground requests. Instead, there is a significant amount of idle disk head time that could be used for freeblock requests, albeit not without some effect on foreground response times. Our study here focuses strictly on use of free bandwidth. As the foreground load increases, opportunities to service freeblock requests are more plentiful, increasing data mining throughput to about 4.9 MB/s (21% of the Atlas 10K's 23MB/s full potential bandwidth). This represents a $7\times$ increase in useful disk head utilization, from 3% to 24%, and it allows the data mining application to complete over 47 full "scans per day" [24] of this 9GB disk with no effect on foreground OLTP performance.

However, as shown in Figure 8b, freeblock scheduling realizes only half of the potential free bandwidth for this environment. As shown in Figure 8c, 18% of the remaining potential is lost to extra seek time, which occurs when pending freeblock requests only exist on a third track (other than the previous and current foreground request). The remaining 28% continues to be rotational latency, either as part of freeblock requests or because no freeblock request could be serviced within the available slot.

Figure 9 helps to explain why only half of the potential free bandwidth is realized for data mining. Specifically, it shows data mining progress and per-OLTP-request breakdown as functions of the time spent on a given disk scan. The main insight here is that the efficiency of freeblock scheduling (i.e., achieved free bandwidth divided by potential free bandwidth) drops steadily as the set of still-desired

background blocks shrinks. As the freeblock scheduler has more difficulty finding conveniently-located freeblock requests, it must look further and further from the previous and current foreground requests. As shown in Figure 9c, this causes extra seek times to increase. Unused rotational latency also increases as freeblock requests begin to incur some latency and as increasing numbers of foreground rotational latencies are found to be too small to allow any pending freeblock request to be serviced. As a result, servicing the last few freeblock requests of a full scan takes a long time; for example, the last 5% of the freeblock requests take 30% of the total time for a scan.

One solution to this problem would be to increase the priority of the last few freeblock requests, with a corresponding impact on foreground requests. The challenge would be to find an appropriate trade-off between impact on the foreground and improved background performance.

An alternate solution would be to take advantage of the statistical nature of many data mining queries. Statistical sampling has been shown to provide accurate results for many queries and internal database operations after accessing only a (randomly-selected) subset of the total dataset [43, 13]. Figure 10 shows the impact of such statistical data mining as a function of the percentage of the dataset needed; that is, the freeblock request is aborted when enough of the dataset has been mined. Assuming that freeblock scheduling within the foreground OLTP workload results in suf-

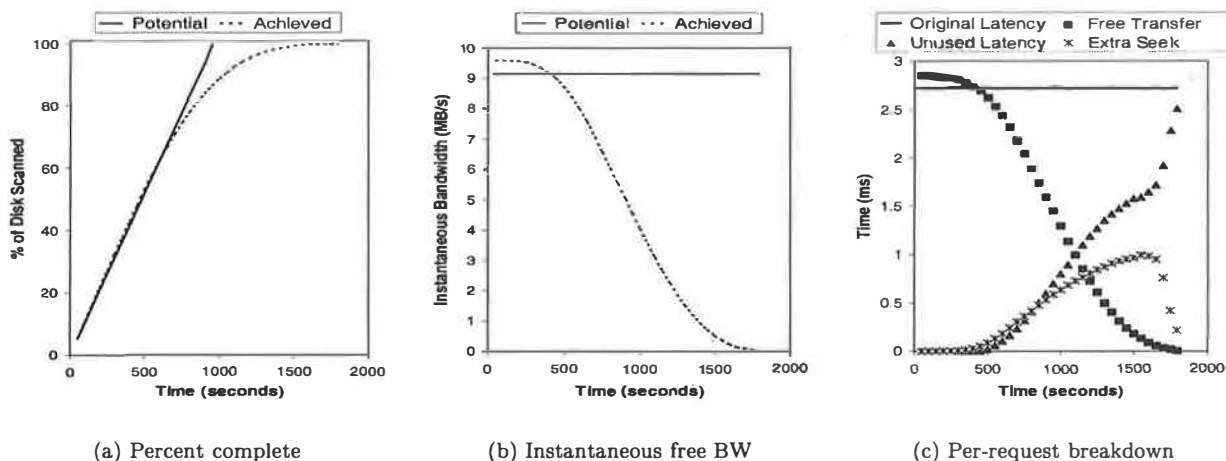


Figure 9: Freeblock-based data mining progress for MPL 7. (a) shows the potential and achieved scan progress. (b) shows the corresponding instantaneous free bandwidth. (c) shows the usage of potential free bandwidth (i.e., original OLTP rotational latency), partitioning it into free transfer time, extra seek time, and unused latency. As expected, the shape of the Free Transfer line in (c) matches that of the achieved instantaneous mining bandwidth in (b). Both exceed the potential free bandwidth early in the scan because many foreground OLTP transfers can also be used by the freeblock scheduler for mining requests when most blocks are still needed.

ficiently random data selection or that the sampling algorithm is adaptive to sampling biases [13], sampling can significantly increase freeblock scheduler efficiency. When any 95% of the dataset is sufficient, efficiency is 40% higher than for full disk scans. For 80% of the dataset, efficiency is at 90% and data mining queries can complete over 90 samples of the dataset per day.

7 Related Work

System designers have long struggled with disk performance, developing many approaches to reduce mechanical positioning overheads and to amortize these overheads over large media transfers. When effective, all of these approaches increase disk head utilization for foreground workloads and thereby reduce the need for and benefits of freeblock scheduling; none have yet eliminated disk performance as a problem. The remainder of this section discusses work specifically related to extraction and use of free bandwidth.

The characteristics of background workloads that can most easily utilize free bandwidth are much like those that can be expressed well with dynamic set [55] and disk-directed I/O [35] interfaces. Specifically, these interfaces were devised to allow application writers to expose order-independent access patterns to storage systems. Application-hinted prefetching interfaces [9, 44] share some of these

same qualities. Such interfaces may also be appropriate for specifying background activities to freeblock schedulers.

Use of idle time to handle background activities is a long-standing practice in computer systems. A subset of the many examples, together with a taxonomy of idle time detection algorithms, can be found in [23]. Freeblock scheduling complements exploitation of idle time. It also enjoys two superior qualities: (1) ability to make forward progress during busy periods and (2) ability to make progress with no impact on foreground disk access times. Starting a disk request during idle time can increase the response time of subsequent foreground requests, by making them wait or by moving the disk head.

In their exploration of write caching policies, Biswas, et al., evaluate a free prewriting mechanism called *piggybacking* [6]. Although piggybacking only considers blocks on the destination track or cylinder, they found that most write-backs could be completed for free across a range of workloads and cache sizes. Relative to their work, our work generalizes both the freeblock scheduling algorithm and the uses for free bandwidth.

Freeblock scheduling relies heavily on the ability to accurately predict mechanical positioning delays (both seek times and rotational latencies). The firmware of most high-end disk drives now supports Shortest-Positioning-Time-First (SPTF) scheduling algorithms, which require similar predictions. Based

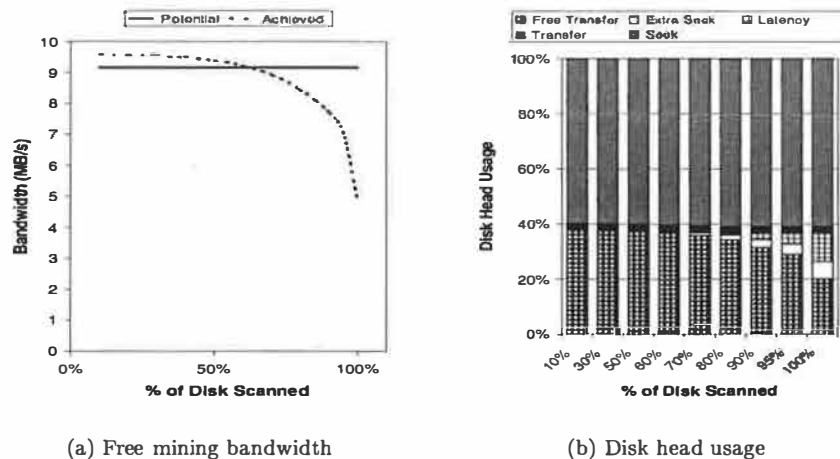


Figure 10: **Freeblock-based data mining performance for statistical queries.** Here, it is assumed that any X% of the disk's data satisfy the needs of a query scan. Below 60%, achieved free bandwidth exceeds potential free bandwidth because of the ability to satisfy freeblock requests from foreground transfers

on this fact, we are confident that freeblock scheduling is feasible. However, it remains to be seen whether freeblock scheduling can be effective outside of disk drive firmware, where complete knowledge of current state and internal algorithms is available.

Freeblock scheduling resembles advanced disk schedulers for environments with a mixed workload of real-time and non-real-time activities. While early real-time disk schedulers gave strict priority to real-time requests, more recent schedulers try to use slack in deadlines to service non-real-time requests without causing the deadlines to be missed [53, 41, 5, 8]. Freeblock scheduling relates to conventional priority-based disk scheduling (e.g., [10, 21]) roughly as modern real-time schedulers relate to their predecessors. However, since non-real-time requests have no notion of deadline slack, freeblock scheduling must be able to service background requests without extending the access latencies of foreground requests at all. Previous disk scheduler architectures would not do this well for non-periodic foreground workloads, such as those explored in this paper.

While freeblock scheduling can provide free media bandwidth, use of such bandwidth also requires some CPU, memory, and bus resources. One approach to addressing these needs is to augment disk drives with extra resources and extend disk firmware with application-specific functionality [1, 32, 46]. Potentially, such resources could turn free bandwidth into free functionality; Riedel, et al., [45] argue exactly this case for the data mining example of Section 6.

Another interesting use of accurate access time pre-

dictions and layout information is *eager writing*, or remapping new versions of disk blocks to free locations very near the disk head [27, 18, 40, 57]. We believe that eager writing and freeblock scheduling are strongly complementary concepts. Although eager writing decreases available free bandwidth during writes by eliminating many seek and rotational delays, it does not do so for reads. Further, eager writing could be combined with freeblock scheduling when using a write-back cache. Finally, as with the LFS cleaning example in Section 5, free bandwidth represents an excellent resource for cleaning and reorganization enhancements to the base eager writing approach [57].

8 Conclusions

This paper describes freeblock scheduling, quantifies its potential under various conditions, and demonstrates its value for two specific application environments. By servicing background requests in the context of mechanical positioning for normal foreground requests, 20–50% of a disk's potential media bandwidth can be obtained with no impact on the original requests. Using simulation, this paper shows that this free bandwidth can be used to clean LFS segments on busy file servers and to mine data on active transaction processing systems.

These results indicate significant promise, but additional experience is needed to refine and realize freeblock scheduling in practice. For example, it remains to be seen whether freeblock scheduling can be implemented outside of modern disk drives, given

their high-level interfaces and complex firmware algorithms. Even inside disk firmware, freeblock scheduling will need to conservatively deal with seek and settle time variability, which may reduce its effectiveness. More advanced freeblock scheduling algorithms will also be needed to deal with request fragmentation, starvation, and priority mixes.

Acknowledgments

We thank Peter Druschel, Garth Gibson, Andy Klosterman, David Petrou, Jay Wylie, Ted Wong and the anonymous reviewers for helping us to refine this paper. We thank the members and companies of the Parallel Data Consortium (including CLARiiON, EMC, HP, Hitachi, Infineon, Intel, LSI Logic, MTI, Novell, PANASAS, Procom, Quantum, Seagate, Sun, Veritas, and 3Com) for their interest, insights, and support. We also thank IBM Corporation for supporting our research efforts. This work was also partially funded by the National Science Foundation via CMU's Data Storage Systems Center.

References

- [1] A. Acharya, M. Uysal, and J. Saltz. Active disks: programming model, algorithms and evaluation. *Architectural Support for Programming Languages and Operating Systems* (San Jose, California), pages 81–91. ACM, 3–7 October 1998.
- [2] R. Agrawal and J. Schafer. Parallel Mining of Association Rules. *IEEE Transactions on Knowledge and Data Engineering*, 8(6), December 1996.
- [3] S. Akyürek and K. Salem. Adaptive block rearrangement. *ACM Transactions on Computer Systems*, 13(2):89–121, May 1995.
- [4] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-volatile memory for fast, reliable file systems. *Architectural Support for Programming Languages and Operating Systems* (Boston, MA, 12–15 October 1992). Published as *Computer Architecture News*, 20(special issue):10–22, October 1992.
- [5] Paul R. Barham. A Fresh Approach to File System Quality of Service. *IEEE 7th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, 1997.
- [6] P. Biswas, K. K. Ramakrishnan, and D. Towsley. Trace driven analysis of write caching policies for disks. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 13–23, May 1993.
- [7] T. Blackwell, J. Harris, and M. Seltzer. Heuristic cleaning algorithms in log-structured file systems. *Annual USENIX Technical Conference* (New Orleans), pages 277–288. Usenix Association, 16–20 January 1995.
- [8] John Bruno, Jose Brustoloni, Eran Gabber, Banu Ozden, and Abraham Siberschatz. Disk Scheduling with Quality of Service Guarantees. *IEEE International Conference on Multimedia Computing and Systems*, 1999.
- [9] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, November 1996.
- [10] S. C. Carson and S. Setia. Analysis of the periodic update write policy for disk cache. *IEEE Transactions on Software Engineering*, 18(1):44–54, January 1992.
- [11] V. Cate and T. Gross. Combining the concepts of compression and caching for a two-level filesystem. *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, CA 8–11 April 1991), pages 200–211. ACM, 1991.
- [12] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1):55, 1997.
- [13] S. Chaudhuri, R. Motwani, and V. Narasayya. Random Sampling for Histogram Construction: How much is enough? *SIGMOD Record*, 27(2):436–447, 1998.
- [14] Ann Chervenak, Vivekanand Vellanki, and Zachary Kurmas. Protecting file systems: A survey of backup techniques. *Joint NASA and IEEE Mass Storage Conference*, March 1998.
- [15] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The Logical Disk: a new approach to improving file systems. *ACM Symposium on Operating System Principles* (Asheville, NC), pages 15–28, 5–8 December 1993.
- [16] P. J. Denning. Effects of scheduling on file memory operations. *AFIPS Spring Joint Computer Conference* (Atlantic City, New Jersey, 18–20 April 1967), pages 9–21, April 1967.
- [17] Database of validated disk parameters for DiskSim. <http://www.ece.cmu.edu/~ganger/disksim/diskspecs.html>.
- [18] R. M. English and A. A. Stepanov. Loge: a self-organizing storage device. *Winter USENIX Technical Conference* (San Francisco, CA), pages 237–251. Usenix, 20–24 January 1992.
- [19] Ussama Fayyad. Taming the Giants and the Monsters: Mining Large Databases for Nuggets of Knowledge. *Databases Programming and Design*, March 1998.
- [20] G. R. Ganger, B. L. Worthington, and Y. N. Patt. *The DiskSim Simulation Environment Version 1.0 Reference Manual*, CSE-TR-358–98. Department of Computer Science and Engineering, University of Michigan, February 1998.
- [21] Gregory R. Ganger and Yale N. Patt. Using System-Level Models to Evaluate I/O Subsystem Designs. *IEEE Transactions Vol. 47 No. 6*, June 1998.
- [22] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Cirang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. *Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, 3–7 October 1998). Published as *SIGPLAN Notices*, 33(11):92–103, November 1998.
- [23] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes. Idleness is not sloth. *Winter USENIX Technical Conference* (New Orleans, LA), pages 201–212. Usenix Association, Berkeley, CA, 16–20 January 1995.
- [24] J. Gray and G. Graefe. Storage metrics. Microsoft Research, Draft of March 1997.
- [25] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. *Summer USENIX Technical Conference* (Boston, June 1994), pages 197–207. USENIX, June 1994.
- [26] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. CURE: An Efficient Clustering Algorithm for Large Databases. *SIGMOD Record*, 27(2):73, 1998.
- [27] R. B. Hagmann. *Low latency logging*. CSL-91–1. Xerox Palo Alto Research Center, CA, February 1991.
- [28] N. G. Hutchinson, S. Manley, M. Federwisch, G. Harris, D. Hitz, S. Kleiman, and S. O'Malley. Logical vs. physical file system backup. *Symposium on Operating Systems Design and Implementation* (New Orleans, LA, 22–25 February 1999), pages 239–249. ACM, February 1999.
- [29] E. H. H. II and R. Finkel. An ASCII database for fast queries of relatively stable data. *Computing Systems*, 4(2):127–155. Usenix Association, Spring 1991.
- [30] D. M. Jacobson and J. Wilkes. *Disk scheduling algorithms based on rotational position*. HPL-CSP-91-7. Hewlett-Packard Laboratories, Palo Alto, CA, 24 February 1991, revised 1 March 1991.
- [31] J. Katcher. *PostMark: a new file system benchmark*. TR3022. Network Appliance, October 1997.
- [32] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for

- intelligent disks (IDISks). *SIGMOD Record*, 27(3):42–52, September 1998.
- [33] G. H. Kim and E. H. Spaff'ord. The design and implementation of Tripwire: a file system integrity checker. *Conference on Computer and Communications Security* (Fairfax, Virginia), pages 18–29, 2–4 November 1994.
- [34] F. Korn, A. Labrinidis, Y. Kotidis, and C. Faloutsos. Ratio Rules: A New Paradigm for Fast, Quantifiable Data Mining. *VLDB*, August 1998.
- [35] D. Kotz. Disk-directed I/O for MIMD multiprocessors. *Symposium on Operating Systems Design and Implementation* (Monterey, CA), pages 61–74. Usenix Association, 14–17 November 1994.
- [36] T. M. Kroeger and D. D. E. Long. The case for efficient file access pattern modeling. *Hot Topics in Operating Systems* (Rio Rico, Arizona), pages 14–19, 29–30 March 1999.
- [37] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the performance of log-structured file systems with adaptive methods. *ACM Symposium on Operating System Principles* (Saint-Malo, France, 5–8 October 1997). Published as *Operating Systems Review*, 31(5):238–252. ACM, 1997.
- [38] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.
- [39] B. McNutt. Background data movement in a log-structured disk subsystem. *IBM Journal of Research and Development*, 38(1):47–58, 1994.
- [40] J. Menon, J. Roche, and J. Kasson. Floating parity and data disk arrays. *Journal of Parallel and Distributed Computing*, 17(1–2):129–139, January–February 1993.
- [41] A. Molano, K. Juvva, and R. Rajkumar. Real-time filesystems. Guaranteeing timing constraints for disk accesses in RT-Mach. *Proceedings Real-Time Systems Symposium* (San Francisco, CA, 2–5 December 1997), pages 155–165. Institute of Electrical and Electronics Engineers Comp. Soc., 1997.
- [42] S. W. Ng. Improving disk performance via latency reduction. *IEEE Transactions on Computers*, 40(1):22–30, January 1991.
- [43] F. Olken and D. Rotem. Simple random sampling from relational databases. *VLDB*, pages 160–169, 1986.
- [44] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. *ACM Symposium on Operating System Principles* (Copper Mountain Resort, CO). Published as *Operating Systems Review*, 29(5):79–95, 3–6 December 1995.
- [45] E. Riedel, C. Faloutsos, G. R. Ganger, and D. F. Nagle. Data mining on an OLTP system (nearly) for free. *ACM SIGMOD Conference 2000* (Dallas, TX), pages 13–21, 14–19 May 2000.
- [46] E. Riedel, G. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia applications. *International Conference on Very Large Databases* (New York, NY, 24–27 August, 1998). Published as *Proceedings VLDB*, pages 62–73. Morgan Kaufmann Publishers Inc., 1998.
- [47] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.
- [48] C. Ruemmler and J. Wilkes. *Disk Shuffling*. HPL-91-156. Hewlett-Packard Laboratories, Palo Alto, CA, October 1991.
- [49] P. H. Seaman, R. A. Lind, and T. L. Wilson. On teleprocessing system design, part IV: an analysis of auxiliary-storage activity. *IBM Systems Journal*, 5(3):158–170, 1966.
- [50] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. *Winter USENIX Technical Conference* (Washington, DC), pages 313–323, 22–26 January 1990.
- [51] M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File system logging versus clustering: a performance comparison. *Annual USENIX Technical Conference* (New Orleans), pages 249–264. Usenix Association, 16–20 January 1995.
- [52] Margo Seltzer. LFS and FFS Supplementary Information, 1995. <http://www.eecs.harvard.edu/~margo/usenix.1995>.
- [53] P. J. Shenoy and H. M. Vin. Cello: a disk scheduling framework for next generation operating systems. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Madison, WI). Published as *Performance Evaluation Review*, 26(1):44–55, June 1998.
- [54] Elizabeth Shriver, Christopher Small, and Keith A. Smith. Why Does File System Prefetching Work? *USENIX Technical Conference* (Monterey, California, June 6–11 1999), pages 71–83. USENIX, 1999.
- [55] D. C. Steere. Exploiting the non-determinism and asynchrony of set iterators to reduce aggregate file I/O latency. *ACM Symposium on Operating System Principles* (Saint-Malo, France, 5–8 October 1997). Published as *Operating Systems Review*, 31(5):252–263. ACM, 1997.
- [56] P. Vongsathorn and S. D. Carson. A system for adaptive disk rearrangement. *Software—Practice and Experience*, 20(3):225–242, March 1990.
- [57] R. Y. Wang, D. A. Patterson, and T. E. Anderson. Virtual log based file systems for a programmable disk. *Symposium on Operating Systems Design and Implementation* (New Orleans, LA, 22–25 February 1999), pages 29–43. ACM, Winter 1998.
- [58] J. Widom. Research Problems in Data Warehousing. *Conference on Information and Knowledge Management*, November 1995.
- [59] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.
- [60] B. L. Worthington, G. R. Ganger, and Y. N. Patt. *Scheduling for modern disk drives and non-random workloads*. CSE-TR-194-94. Department of Computer Science and Engineering, University of Michigan, March 1994.
- [61] Xiang Yu, Benjamin Gum, Yuqun Chen, Randolph Y. Wang, Kai Li, Arvind Krishnamurthy, and Thomas E. Anderson. Trading Capacity for Performance in a Disk Array. *Symposium on Operating Systems Design and Implementation*, October 2000.
- [62] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: A New Data Clustering Algorithm and Its Applications. *Data Mining and Knowledge Discovery*, 2(1), 1997.

Latency Management in Storage Systems

Rodney Van Meter*

Quantum Corp.

rdv@alumni.caltech.edu

Minxi Gao

U.C. Berkeley

minxi@eecs.berkeley.edu

Abstract

Storage Latency Estimation Descriptors, or SLEDs, are an API that allow applications to understand and take advantage of the dynamic state of a storage system. By accessing data in the file system cache or high-speed storage first, total I/O workloads can be reduced and performance improved. SLEDs report estimated data latency, allowing users, system utilities, and scripts to make file access decisions based on those retrieval time estimates. SLEDs thus can be used to improve individual application performance, reduce system workloads, and improve the user experience with more predictable behavior.

We have modified the Linux 2.2 kernel to support SLEDs, and several Unix utilities and astronomical applications have been modified to use them. As a result, execution times of the Unix utilities when data file sizes exceed the size of the file system buffer cache have been reduced from 50% up to more than an order of magnitude. The astronomical applications incurred 30-50% fewer page faults and reductions in execution time of 10-35%. Performance of applications which use SLEDs also degrade more gracefully as data file size grows.

1 Introduction

Storage Latency Estimation Descriptors, or SLEDs, abstract the basic characteristics of data retrieval in a device-independent fashion. The ultimate goal is to create a mechanism that reports detailed performance characteristics without being tied to a particular technology.

Storage systems consist of multiple devices with different performance characteristics, such as RAM (e.g., the operating system's file system buffer cache), hard disks, CD-ROMs, and magnetic tapes. These devices may be attached to the machine on which the application is running, or may be attached to a separate server machine. All of these elements communicate via a variety of interconnects, including SCSI buses and ethernet. As systems and applications create and access data, it moves among the various devices along these interconnects.

Hierarchical storage management (HSM) systems with capacities up to a petabyte currently exist, and systems up to 100PB are currently being designed [LLJR99, Shi98]. In such large systems, tape will continue to play an important role. Data is migrated to tape for long-term storage and fetched to disk as needed, analogous to movement between disk and RAM in conventional file systems. A CD jukebox or tape library automatically mounts media to retrieve requested data.

Storage systems have a significant amount of dynamic state, a result of the history of accesses to the system. Disks have head and rotational positions, tape drives have seek positions, autochangers have physical positions as well as a set of tapes mounted on various drives. File systems are often tuned to give cache priority to recently used data, as a heuristic for improving future accesses. As a result of this dynamic state, the latency and bandwidth of access to data can vary dramatically; in disk-based file systems, by four orders of magnitude (from microseconds for cached, unmapped data pages, to tens of milliseconds for data retrieved from disk), in HSM systems, by as much as eleven (microseconds up to hundreds of seconds for tape mount and seek).

File system interfaces are generally built to hide this variability in latency. A `read()` system call works the same for data to be read from the file system buffer cache

* Author's current address: Nokia, Santa Cruz, CA.

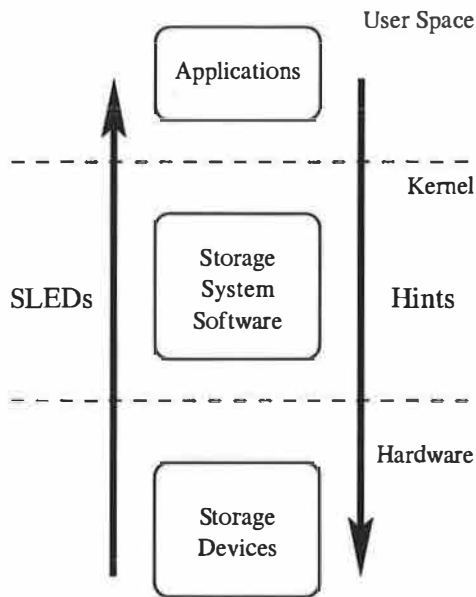


Figure 1: SLEDs and hints in the storage system stack

as for data to be read from disk. Only the behavior is different; the semantics are the same, but in the first case the data is obtained in microseconds, and in the second, in tens of milliseconds.

CPU performance is improving faster than storage device performance. It therefore becomes attractive to expend CPU instructions to make more intelligent decisions concerning I/O. However, with the strong abstraction of file system interfaces, applications are limited in their ability to contribute to I/O decisions; only the system has the information necessary to schedule I/Os.

SLEDs are an API that allows applications and libraries to understand both the dynamic state of the storage system and some elements of the physical characteristics of the devices involved, in a device-independent fashion. Using SLEDs, applications can manage their patterns of I/O calls appropriately. They may reorder or choose not to execute some I/O operations. They may also report predicted performance to users or other applications.

SLEDs can be contrasted to file system hints, as shown in Figure 1. Hints are the flow of information down the storage system stack, while SLEDs are the flow information up the stack. The figure is drawn with the storage devices as well as the storage system software participating. In current implementations of these concepts, the storage devices are purely passive, although their characteristics are measured and presented by proxy for SLEDs.

This paper presents the first implementation and measurement of the concept of SLEDs, which we proposed in an earlier paper [Van98]. We have implemented the SLEDs system in kernel and library code under Linux (Red Hat 6.0 and 6.1 with 2.2 kernels), and modified several applications to use SLEDs.

The applications we have modified demonstrate the different uses of SLEDs. `wc` and `grep` were adapted to reorder their I/O calls based on SLEDs information. The performance of `wc` and `grep` have been improved by 50% or more over a broad range of file sizes, and more than an order of magnitude under some conditions. `find` is capable of intelligently choosing not to perform certain I/Os. The GUI file manager `gmc` reports estimated retrieval times, improving the quality of information users have about the system.

We also modified LHEASOFT, a large, complex suite of applications used by professional astronomers for image processing [NAS00]. One member of the suite, `fimhisto`, which copies the data file and appends a histogram of the data to the file, showed a reduction in page faults of 30-50% and a 15-25% reduction in execution time for files larger than the file system buffer cache. `fimgbin`, which rebins an image, showed a reduction of 11-35% in execution time for various parameters. The smaller improvements are due in part to the complexity of the applications, relative to `wc` and `grep`.

The next section presents related work. This is followed by the SLEDs design, then details of the implementation, and results. The paper ends with future work and conclusions.

2 Related Work

The history of computer systems has generally pushed toward increasingly abstract interfaces hiding more of the state details from applications. In a few instances, HSM systems provide some ability for HSM-aware applications to determine if files are online (on disk) or offline (on tape or other low levels of the hierarchy). Microsoft's Windows 2000 (formerly NT5) [v199], TOPS-20, and the RASH system [HP89] all provide or provided a single bit that indicates whether the file is online or offline. SLEDs extends this basic concept by providing more detailed information.

Steere's file sets [Ste97] exploit the file system cache on a file granularity, ordering access to a group of files to

present the cached files first. However, there is no notion of intra-file access ordering.

Some systems provide more direct control over what pages are selected for prefetching or for cache replacement. Examples include TOPS-10 [Dig76] and TOPS-20 and Mach's user-definable pagers, Cao's application-controlled file caching [CFKL96], and the High Performance Storage System (HPSS) [WC95].

Still other systems have improved system performance by a mechanism known as *hints*. Hints are flow of information from the application to the system about expected access orders and data reuse. They are, in effect, the inverse of SLEDs, in which information flows from the system to the application. Hints may allow the system to behave more efficiently, but do not allow the application to participate directly in I/O decisions, and cannot report expected I/O completion times to the application or user. Good improvements have been reported with hints over a broad range of applications [PGG⁺95]. Reductions in elapsed time of 20 to 35 percent for a single-disk system were demonstrated, and as much as 75 percent for ten-disk systems. Hints cannot be used across program invocations, or take advantage of state left behind by previous applications. However, hints can help the system make more intelligent choices about what data should be kept in cache as an application runs.

Hillyer and Silberschatz developed a detailed device model for a DLT tape drive that allows applications to schedule I/Os effectively [HS96a, HS96b]. Sandst  and Midstraum extended their model, simplifying it and making it easier to use [SM99]. The goal is the same as SLEDs, effective application-level access ordering, but is achieved in a technology-aware manner. Such algorithms are good candidates to be incorporated into SLEDs libraries, hiding the details of the tape drive from application writers.

For disk drives, detailed models such as Ruemmler's [RW94] and scheduling work such as Worthington's [WGP94] may be used to enhance the accuracy of SLEDs.

Real-time file systems for multimedia, such as Anderson's continuous media file system [AOG92] and SGI's XFS [SDH⁺96], take reservation requests and respond with an acceptance or rejection. SLEDs could be integrated with such systems to provide substrate (storage and transfer subsystems communicating their characteristics to the file system to improve its decision-making capabilities), or to increase the usefulness of the information provided to applications about their requests.

Such systems calculate information similar to SLEDs internally, but currently do not expose it to applications, where it could be useful.

Distributed storage systems, such as Andrew and Coda [Sat90], Tiger [BFD97], Petal [LT96], and xFS [ADN⁺95], present especially challenging problems in representing performance data, as many factors, including network characteristics and local caching, come into play. We propose that SLEDs be the vocabulary of communication between clients and servers as well as between applications and operating systems.

Mobile systems, including PDAs and cellular phones, are an especially important area where optimizing for latency and bandwidth are important [FGBA96]. Perhaps the work most like SLEDs is Odyssey [NSN⁺97]. Applications make resource reservations with SLEDs-like requests, and register callbacks which can return a new value for a parameter such as network latency or bandwidth as system conditions change.

Attempts to improve file system performance through more intelligent caching and prefetching choices include Kroeger's [Kro00] and Griffioen's [GA95]. Both use file access histories to predict future access patterns so that the kernel can prefetch more effectively. Kroeger reports I/O wait time reductions of 33 to 90 percent under various conditions on his implementation, also done on a Linux 2.2.12 kernel.

Kotz has simulated and studied a mechanism called *disk-directed I/O* for parallel file systems [Kot94]. Compute processors (CPs) do not adjust their behavior depending on the state of the I/O system, but collectively aggregate requests to I/O processors (IOPs). This allows the I/O processors to work with deep request queues, sorting for efficient access to achieve a high percentage of the disk bandwidth. Unlike SLEDs, the total I/O load is not reduced by taking dynamic advantage of the state of client (CP) caches, though servers (IOPs) may gain a similar advantage by ordering already-cached data to be delivered first.

Asynchronous I/O, such as that provided by POSIX I/O or VMS, can also reduce application wait times by overlapping execution with I/O. In theory, posting asynchronous read requests for the entire file, and processing them as they arrive, would allow behavior similar to SLEDs. This would need to be coupled with a system-assigned buffer address scheme such as containers [PA94], since allocating enough buffers for files larger than memory would result in significant virtual memory thrashing.

```

struct sled {
    long offset; /* into the file */
    long length; /* of the segment */
    float latency; /* in seconds */
    float bandwidth; /* in bytes/sec */
};

```

Figure 2: SLED structure

3 SLEDs Design

The basic elements of the Storage Latency Estimation Descriptor structure are shown in Figure 2. SLEDs represent the estimated latency to retrieve specific data elements of a file. A SLED consists of the byte offset within the file, the length in bytes of this section, and the performance estimates. The estimates are the latency to the first byte of the section and the bandwidth at which data will arrive once it has begun. Floating point numbers are used to represent the latency because the necessary range exceeds that of a normal integer. We chose floating point numbers for bandwidth for consistency of representation and ease of arithmetic.

Different sections (usually blocks) of a file may have different retrieval characteristics, and so will be represented by separate SLEDs. For large files, as a file is used and reused, the state of a file may ultimately be represented by a hundred or more SLEDs. Moving from the beginning of the file to the end, each discontinuity in storage media, latency, or bandwidth results in another SLED in the representation.

Applications take advantage of the information in a SLED in one of three possible ways: reordering I/Os, pruning I/Os, or reporting latencies. Each is detailed in the following subsections.

3.1 Reordering I/Os

By accessing the data currently in primary memory first, and items that must be retrieved from secondary or tertiary storage later, the number of physical I/Os that must be performed may be reduced. This may require algorithmic changes in applications.

Figure 3 shows how two linear passes across a file behave with normal LRU caching when the file is larger than the cache size. A five-block file is accessed using a cache which is only three blocks. The contents of the

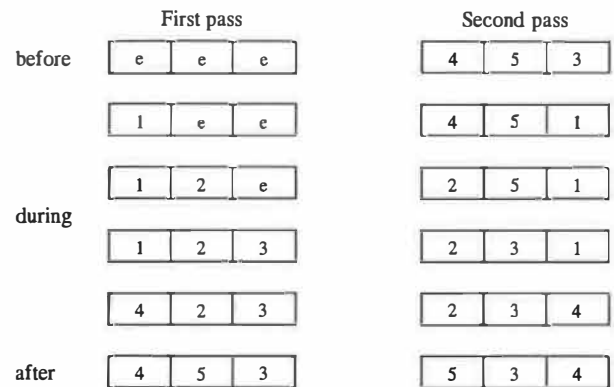


Figure 3: Movement of data among storage levels during two linear passes

cache are shown as file block numbers, with “e” being empty. The second pass gains no advantage from the data cached as a result of the first pass, as the tail of the file is progressively thrown out of cache during the current pass. The two passes may be within a single application, or two separate applications run consecutively. Behavior is similar whether the two levels are memory and disk, as in a normal file system, or disk and tape, as in any hierarchical storage management system which caches partial files from tape to disk.

By using SLEDs, the second pass would be able to recognize that reading the tail of the file first is advantageous. In this case, blocks 3, 4, and 5 would be known to be cached, and read before blocks 1 and 2. The total number of blocks retrieved from the lower storage level in this second pass would only be two instead of five.

3.2 Pruning I/Os

By algorithmically choosing not to execute particular I/Os, an application may improve its performance by orders of magnitude, as well as be a better citizen by reducing system load.

A simple example that combines both pruning and reordering is an application which is looking for a specific record in a large file or set of files. If the desired record’s position is toward the end of the data set as normally ordered, but already resides in memory rather than on disk or tape (possibly as a result of recent creation or recent access), it may be among the first accessed. As a result, the application may terminate without requesting data which must be retrieved for disk or tape, and performance may improve by an order of magnitude or more.

It may also simply be desirable to run an application with the minimum number of I/O operations, even at the cost of reduced functionality. This may be applied in, for example, environments that charge per I/O operation, as used to be common for timesharing systems.

3.3 Reporting Latency

Applications in several categories depend on or can be improved by an ability to predict their performance. Quality of service with some real-time component is the most obvious but not the only such category; any I/O-intensive application on which a user depends for interactive response is a good candidate to use SLEDs.

Systems that provide quality of service guarantees generally do so with a reservation mechanism, in which applications request a specific performance level, and the system responds with a simple yes or no about its ability to provide the requested performance. Once the reservation has begun, QoS systems rarely provide any additional information about the arrival of impending data.

Most applications which users interact with directly are occasionally forced to retrieve significant amounts of data, resulting in the appearance of icons informing the user that she must wait, but with no indication of the expected duration. Better systems (including web browsers) provide visible progress indicators. Those indicators are generally estimated based on partial retrieval of the data, and so reflect current system conditions, but cannot be calculated until the data transfer has begun. Dynamically calculated estimates can be heavily skewed by high initial latency, such as in an HSM system. Using SLEDs instead provides a clearer picture of the relationship of the latency and bandwidth, providing complementary data to the dynamic estimate, and can be provided before the retrieval operation is initiated.

Both types of applications above have a common need for a mechanism to communicate predicted latency of I/O operations from storage devices to operating systems to libraries to applications. SLEDs is one proposal for the vocabulary of such communication.

3.4 Design Limitations

SLEDs, as currently implemented, describe the state of the storage system at a particular instant. This state, however, varies over time. Mechanical positioning of

devices changes, and cached data can change as a result of I/Os performed by the application, other applications or system services, or even other clients in a distributed system. Adding a lock or reservation mechanism would improve the accuracy and lifetime of SLEDs by controlling access to the affected resources.

Another possibility is to include in the SLEDs themselves some description of how the system state will change over time, such as a program segment that applications could use to predict which pages of a file would be flushed from cache based on current page replacement algorithms.

4 Implementation

The implementation of SLEDs includes some kernel code to assess and report the state of data for an open file descriptor, an `ioctl` call for communicating that information to the application level, and a library that applications can use to simplify the job of ordering I/O requests based on that information.

This section describes the internal details of the implementation of the kernel code, library and application modifications.

4.1 Kernel

We modified the Linux kernel to determine which device the pages for a file reside on, and whether or not the pages are currently in memory. All of the changes were made in the virtual file system (VFS) layer, independent of the on-disk data structure of ext2 or ISO9660.

A `sleds_table`, kept in the kernel, is filled by calling a script from `/etc/rc.d/init.d` every time the machine is booted. The `sleds_table` has a latency and bandwidth entry for every storage device, as well as NFS-mounted partitions and primary memory. The latency and bandwidth for both local and network file systems are obtained by running the `lmbench` benchmark [MS96]. The current implementation keeps only a single entry per device; for better accuracy, entries which account for the different bandwidths of different disk zones will be added in a future version [Van97]. The script fills the kernel table via a new `ioctl` call, `FSLEDS_FILL`, added to the generic file system `ioctl`.

Applications can retrieve the SLEDs for a file using another new `ioctl` call, `FSLEDS_GET`, which returns a vector of SLEDs. To build the vector of SLEDs and their latency and bandwidth, each virtual memory page of the data file is checked. After the kernel finds out where a page of the data file resides, it assigns a latency and bandwidth from the `sleds_table` to this page. If consecutive pages have the same latency and bandwidth, i.e., they are in the same storage device, they are grouped into one SLED. During this process, the length and offset of the SLEDs are also assigned.

4.2 Library and API

The means of communication between application space and kernel space is via `ioctl` calls which return only SLEDs. This form is not directly very useful, so a library was also written that provides additional services. The library provides a routine to estimate delivery time for the entire file, and several routines to help applications order their I/O requests efficiently.

The three primary library routines for reordering I/O are `sleds_pick_init`, `sleds_pick_next_read`, and `sleds_pick_finish`. Applications first open the file, then call `sleds_pick_init`, which uses `ioctl` to retrieve the SLEDs from the kernel. `sleds_pick_next_read` is called repeatedly to advise the application where to read from the file next. The application then moves to the recommended position via `lseek`, and calls `read` to retrieve the next chunk of the file. The preferred size of the chunks the application wants to read is specified as an argument to `sleds_pick_init`, and `sleds_pick_next_read` will return chunks that size or smaller. The application is presumed to be following the library's advice, but it does not check. The library will return each chunk of the file exactly once.

The library checks for the lowest latency among unseen chunks, then chooses to return the chunk with the lowest file offset among those with equivalent latencies. In the simple case of a disk-based file system with a cold cache, this algorithm will degenerate to linear access of the file. As currently implemented, the SLEDs are retrieved from the kernel only when `sleds_pick_init` is called. Refreshing the state of those SLEDs occasionally would allow the library to take advantage of any changes in state caused by e.g. file prefetching.

A library routine, `sleds_total_delivery_time`, provides an estimate of the amount of time required

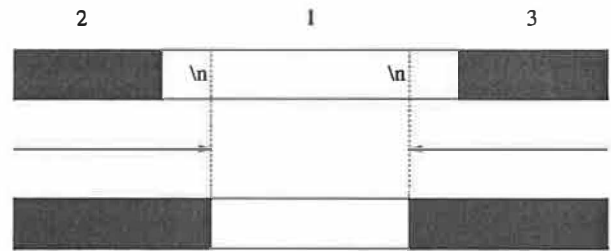


Figure 4: Adjusting SLEDs for record boundaries

to read the entire file, for applications only interested in reporting or using that value. It takes an argument, `attack_plan`, which currently can be either `SLEDs_LINEAR` or `SLEDs_BEST`, to describe the intended access pattern.

Generally, the library and application are most efficient if these accesses can be done on page boundaries. However, many applications are interested in variable-sized records, such as lines of text. An argument to `sleds_pick_init` allows the application to ask for record-oriented SLEDs, and to specify the character used to identify record boundaries.

The library prevents applications from running over the edge of a low-latency SLED and causing data to be fetched from higher-latency storage when in record-oriented mode. It does this by pulling in the edges of the SLEDs from page boundaries to record boundaries, as shown in Figure 4. The leading and trailing record fragments are pushed out to the neighboring SLEDs, which are higher latency. This requires the SLEDs library to perform some I/O itself to find the record boundaries. In the figure, the gray areas are high-latency SLEDs in a file, and the white area is a low-latency SLED. The numbers above represent the access ordering assigned by the library. The upper line is before adjustment for record boundaries, and the lower line is after adjustment for variable-sized records with a linefeed record separator.

4.3 Applications

Figure 5 shows pseudocode for an application using the SLEDs pick library. After initializing the SLEDs library, the application loops, first asking the library to select a file offset, then seeking to that location and reading the amount recommended.

Applications that have been modified to use SLEDs include the GNU versions of the Unix system utilities `wc`,

function	arguments	return value
sleds_pick_init	file descriptor, preferred buffer size	buffer size
sleds_pick_next_read	file descriptor, buffer size, record flag	read location, size
sleds_pick_finish	file descriptor	(none)
sleds_total_delivery_time	file descriptor, attack plan	estimated delivery time

Table 1: SLEDs library routines

```

int offset, nbytes, Remain;
int FileSize, fd;
char buffer[BUFSIZE];

fd = open(FileName, flags);
sleds_pick_init(fd, BUFSIZE);

for( Remain = FileSize ; Remain ;
    Remain -= nbytes ){
    nelem = MIN(Remain, BUFSIZE);
    sleds_pick_next_read(fd, &offset,
                        &nbytes);

    lseek(fd, offset, SEEK_SET);
    read(fd, buffer, nbytes);
    process_data(buffer, nbytes);
}
sleds_pick_finish(fd);
close(fd);

```

Figure 5: Application pseudocode

grep, and find, and the GNOME file manager gmc. These examples demonstrate the three ways in which SLEDs can be useful. The first two use SLEDs to reorder I/O operations, gaining performance and reducing total I/O operations by taking advantage of cached data first, using algorithms similar to figure 5. find is modified to include a predicate which allows the user to select files based on total estimated latency (either greater than or less than a specified value). This can be used to prune expensive I/O operations. gmc reports expected file retrieval times to the user, allowing him or her to choose whether or not to access the file.

We have also adapted two members of the LHEASOFT suite of applications, fimhisto and fimgbn, to use SLEDs. NASA's Goddard Space Flight Center supports LHEASOFT, which provides numerous utilities for the processing of images in the Flexible Image Transport System, or FITS, format used by professional astronomers. The FITS format includes image metadata, as well as the data itself.

4.4 Implementation Limitations

The current implementation provides only a basic estimate of latency based on device characteristics, with no indication of current head or rotational position. The primary information provided is a distinction between levels of the storage system, with estimates of the bandwidth and latency to retrieve data at each level. This information is effective for disk drives, but will need to be updated for tape drives. Future extensions are expected to provide more detailed mechanical estimates.

5 Results and Analysis

The benefits of SLEDs include both useful predictability in I/O execution times, and improvements in execution times for those applications which can reorder their I/O requests. In this section we discuss primarily the latter, objectively measurable aspects.

We hypothesize that reordering I/O requests according to SLEDs information will reduce the number of hard page faults, that this will translate directly to decreased execution times, and that the effort required to adopt SLEDs is reasonable. To validate these hypotheses, we measured both page faults and elapsed time for the modified applications described above, and report the number of lines of code changed.

SLEDs are expected to benefit hierarchical storage management systems, with their very high latencies, more than other types of file systems. The experiments shown here are for normal on-disk file system structures cached in primary memory. Thus, the results here can be viewed as indicative of the positive benefits of SLEDs, but gains may be much greater with HSM systems.

5.1 Experimental Setup

To measure the effect of reordering data requests, the average time and page faults taken to execute the applications with SLEDs were plotted against the values without SLEDs. We used the system time command to do the measurements. Tests were run on `wc` and `grep` for data files that reside on hard disk, CD-ROM, and NFS file systems. `grep` was tested in two different modes, once doing a full pass across the data file, and once terminating on the first match (using the `-q` switch). The modified LHEASOFT applications were run only against hard disk file systems.

During the test runs, no other user activity was present on the systems being measured. However, some variability in execution times is unavoidable, due to the physical nature of I/O and the somewhat random nature of page replacement algorithms and background system activity. All runs were done twelve times (representing a couple of days' execution time in total) and 90% confidence intervals calculated. The graphs show the mean and confidence intervals, though in some cases the confidence intervals are too small to see.

Data was taken for test file sizes of 8 to 128 megabytes, in multiples of eight, for most of the experiments. With a primary memory size of 64 MB, this upper bound is twice the size of primary memory and roughly three times the size of the portion of memory available to cache file pages. We expect no surprises in the range above this value, but a gradual decrease in the relative improvement.

Because SLEDs are intended to expose and take advantage of dynamic system state, all experiments were done with a warm file cache. A warm cache is the natural state of the system during use, since files that have been recently read or written are commonly accessed again within a short period of time. SLEDs provide no benefit in the case of a completely cold RAM cache for a disk-based file system. The first run to warm the cache was discarded from the result. The runs were done repeatedly in the same mode, so that, for example, the second run of `grep` without SLEDs found the file system buffer cache in the state that the first run had left it.

Tables 2 and 3 contain the device characteristics used during these experiments.

Table 4 lists the number of lines of source code modified in each application. The "src" columns are lines of code in the main application source files. The "lib" are lines

level	latency	throughput
memory	175 nsec	48 MB/s
hard disk	18 msec	9.0 MB/s
CD-ROM	130 msec	2.8 MB/s
NFS	270 msec	1.0 MB/s

Table 2: Storage levels used for measuring Unix utilities

level	latency	throughput
memory	210 nsec	87 MB/s
hard disk	16.5 msec	7.0 MB/s

Table 3: Storage levels used for measuring LHEASOFT utilities

of code in additional, shared, linked-in libraries. The "modified" columns are lines of code added or modified, and the "total" columns are the totals. The LHEASOFT `cfitsio` library modifications are shared, used in both `fimhisto` and `fimgbin`. The `grep` modifications are most extensive because of the need to buffer and sort output in a different fashion.

5.2 Unix Utilities

In `gmc`, a new simple panel is added to the file properties dialog box, as shown in figure 6. This follows closely the implementation of other windows such as the "general" and "URL" properties panels. The SLEDs panel reports the length, offset, latency, and bandwidth of each SLED, as well as the estimated total delivery time for the file. Users can interactively use this panel to decide whether or not to access files; this is expected to be especially useful in HSM systems and low-bandwidth distributed systems. The same approach could be used with a web browser, if HTTP were extended to support SLEDs across the network.

application	src lines of code		lib lines of code	
	modified	total	modified	total
grep	560	1930	-	20K
wc	140	530	-	48K
find	70	1,600	-	23K
gmc	93	1,500	-	180K
cfitsio	-	-	190	101K
fimhisto	49	645	190	260K
fimgbin	45	870	190	260K

Table 4: Lines of code modified

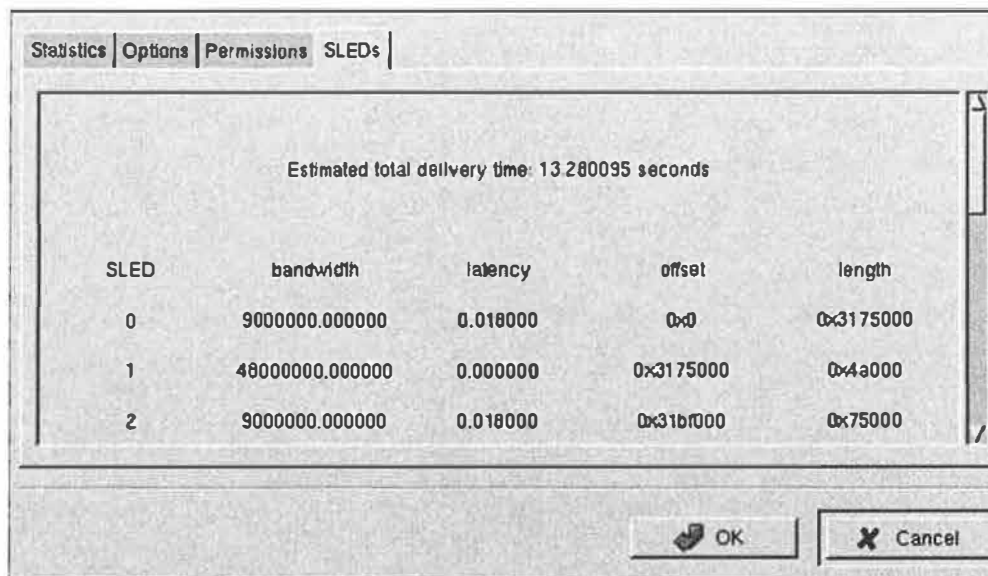


Figure 6: gmc file properties panel with SLEDs

The applications `wc` and `grep` implemented with SLEDs have a switch on the command that allows the user to choose whether or not to use SLEDs. If the SLEDs switch is specified, instead of accessing the data file from the beginning to the end, the application will call `sleds_pick_init`, `sleds_pick_next_read` and `sleds_pick_finish`, and use them as described in section 4.2.

For `wc`, since the order of data access is not significant, little overhead is generated in modifying the code. For applications where the order of data access is influential in code design, such as `grep`, more code changes are needed and as a result may have heavier execution overhead. In our implementation, most of the design with SLEDs is adopted from the one without SLEDs. However, unless the user chooses not to output the matches, the result will have to be output to `stdout` in the order that they appear in the file. To deal with this, we have to store a match in a linked list when traversing the data file in the order recommended by SLEDs. We sort the matches in the end by their offset in the file and then dump them to `stdout`. As a result, switches such as `-A`, `-B`, `-b`, and `-n` had to be reimplemented.

Consider searching a large source tree, such as the Linux kernel. Programmers may do `find -exec grep` (which runs the `grep` text search utility for every file in a directory tree that matches certain criteria, such as ending in `.c`) while looking for a particular routine. If the routine is near the end of the set of files as normally scanned by `find`, or if the user types control-C after

seeing what he wants to see, the entry may be cached but earlier files may already have been flushed. Repeating the operation, then, causes a complete rescan and fetch from high-latency storage. The first author often does exactly this, and the SLEDs-aware `find` allows him to search cache first, then higher latency data only as needed.

Standard `find` provides a switch that stops it from crossing mount points as it searches directory trees. This is useful to, for example, prevent `find` from running on NFS-mounted partitions, where it can overload a server and impair response time for all clients. On HSM systems, users may wish to ignore all tape-resident data, or to read data from a tape currently mounted on a drive, but ignore those that would require mounting a new tape. In wide-area systems, users may wish to ignore large files that must come across the network.

In our modified `find`, the user can choose to find files that have a total delivery time of less than, equal to, or greater than a certain time. `find -latency +n` looks for files with more than n seconds total retrieval time, n means exactly n seconds and `-n` means less than n seconds. `mn` or `Mn` instead of n can be used for units of milliseconds, and `un` or `Un` used for microseconds. The SLEDs library routine `sleds_total_delivery_time` was used for this comparison. Only 2 extra routines (less than 100 lines of code) were needed to add SLEDs to `find` and all functionality has been kept the same. These two extra routines work and were implemented similarly to other

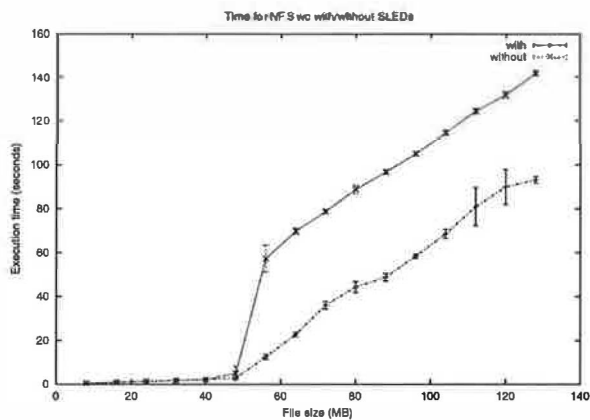


Figure 7: wc times over NFS, with and without SLEDs, warm cache. The legends on the graphs are correct, but somewhat difficult to follow; look first for the plus signs and Xes. The dashed and solid lines in the legend refer to the error bars, not the data lines.

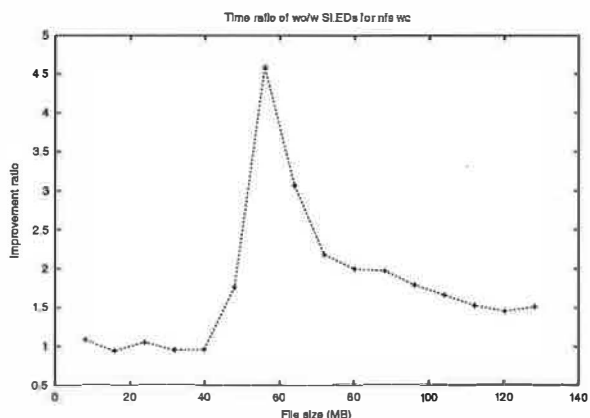


Figure 8: wc time ratios (speedup) over NFS, with and without SLEDs, warm cache

predicates such as `-atime`.

Figure 7 presents the execution times for `wc` against file size on an NFS file system with and without SLEDs. As to be expected, SLEDs starts to show an advantage at file sizes over about 50MB as the cache becomes unable to hold the entire file. The difference in execution time remains about constant afterwards since the average usage of cached data by SLEDs is expected to be determined by the cache size, which is constant. As a result, we have the best percentage gain at around 60MB. We also noticed a very consistent performance gain, as shown by the small error bars in the plot. Figure 8 is the ratio of the two curves in Figure 7. The execution time without SLEDs is divided by the execution time with SLEDs, providing a speedup number. As we can see, this ratio

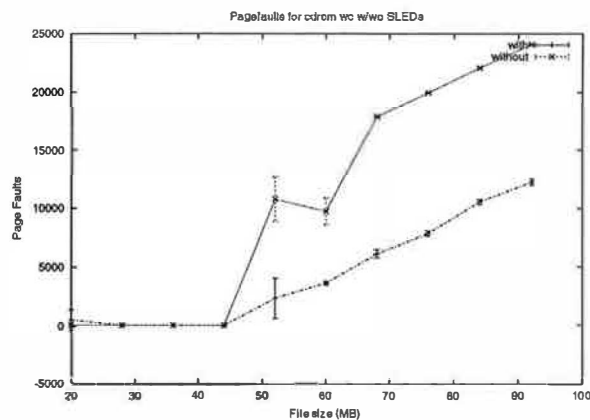


Figure 9: wc page faults on CD-ROM, with and without SLEDs, warm cache

peaks at around 60MB at a value of as large as 4.5, and can be comfortably considered to be a 50% or better improvement over a broad range.

Figure 9 plots the pagefaults for `wc` against file size on cdrom with and without SLEDs. As to be expected, this result shows a close correlation with the execution time. Without SLEDs, both the execution time and pagefaults increase sharply. With SLEDs, the increase in both is gradual.

Figure 10 plots the execution time for `grep` for all matches against file size on cdrom with and without SLEDs. Although there is a small amount of overhead for small files, `grep` also demonstrated a very favorable gain of about 15 seconds for CD-ROM for large files. This can be interpreted as the time taken to fill the file cache from CD when SLEDs are not used, as the application derives essentially no benefit from the cached data in this case.

Because this approach requires buffering matches before output, if the percentage of matches is large, performance can be hurt by causing additional paging. All experiments presented here are for small match percentages (kilobytes out of megabytes) output in the correct order.

The increase in execution time for small files is all CPU time. This is due to the additional complexity of record management with SLEDs, and to more data copying. We used `read()`, rather than `mmap()`, which does not copy the data to meet application alignment criteria. An `mmap`-friendly SLEDs library is feasible, which should reduce the CPU penalty. The increase appears large in percentage terms, but is a small absolute value. Regard-

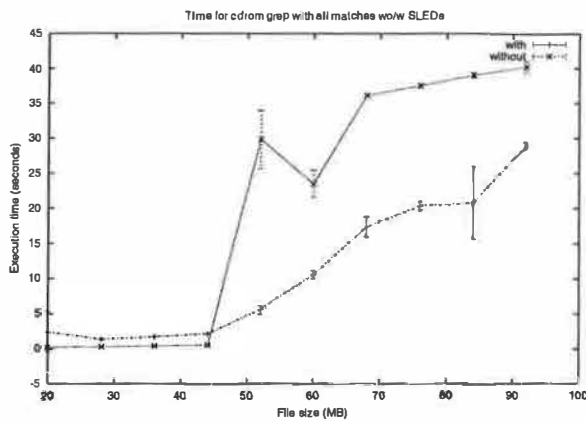


Figure 10: Execution time of `grep` for all matches, CD-ROM, warm cache

less, one of the premises of SLEDs is that modest CPU increases are an acceptable price to pay for reduced I/O loads.

Figure 11 plots the execution time for `grep` for the first match against file size on an ext2 file system (local hard disk) with and without SLEDs, for a single match that was placed randomly in the test file. The first match termination, if it finds a hit on cached data, can run without executing any physical I/O at all. Because the application reads all cached data first when using SLEDs, it has a higher probability of terminating before doing any physical I/O. The non-SLEDs run is often forced to do lots of I/O because it reads from the beginning of the file rather than reading cached data first, regardless of location. Quite dramatic speedups can therefore occur when using SLEDs, relative to a non-SLEDs run. This is the ideal benchmark for SLEDs in terms of individual application performance.

The execution time ratio for `grep` with the first match against file size on the ext2 file system, with and without SLEDs, is shown in Figure 12. In addition, we have computed the cumulative distribution function (CDF) for `grep` for the first match on an NFS file system with and without SLEDs, as shown in Figure 13.

The large error bars in Figure 11 for the case without SLEDs are indicative of high variability in the execution time caused by poor cache performance. The cumulative distribution function for execution times shown in Figure 13 suggests that `grep` without SLEDs gained essentially no benefit from the fact that a majority of the test file is cached.

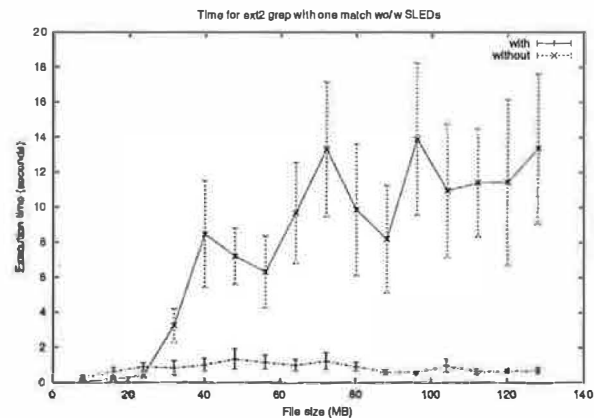


Figure 11: Execution time of `grep` for one match, ext2 FS, warm cache

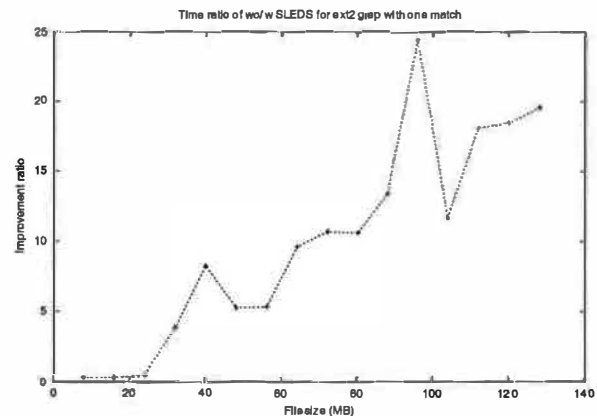


Figure 12: Ratio of mean execution time (speedup) of `grep` for one match, ext2 FS, warm cache, with and without SLEDs

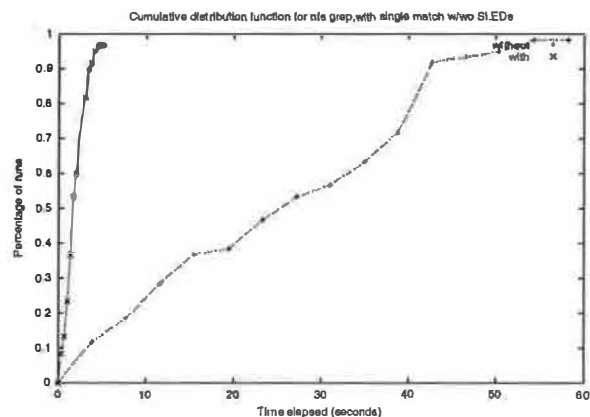


Figure 13: Cumulative distribution of execution time of `grep` for one match, NFS, warm cache, for a 64MB file

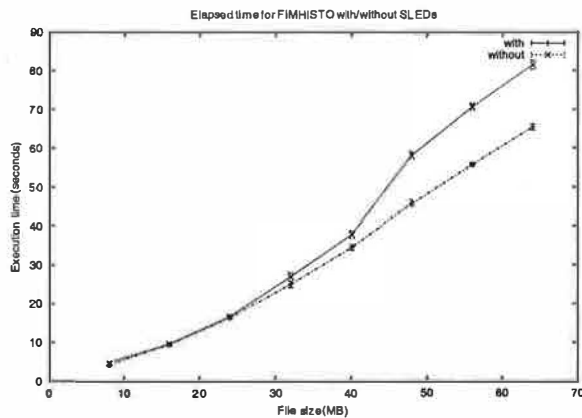


Figure 14: Elapsed time for `fimhisto`, ext2 FS, warm cache

5.3 LHEASOFT

`fimhisto` copies an input data image file to an output file and appends an additional data column containing a histogram of the pixel values. It is implemented in three passes. The first pass copies the main data unit without any processing. The second pass reads the data again (including performing a data format conversion, if necessary) to prepare for binning the data into the histogram. The third pass performs the actual binning operation, then appends the histogram to the output file. This three-pass algorithm resulted in observed cache behavior like that shown in Figure 3.

We adapted `fimhisto` to use SLEDs in the second and third passes over the data, reordering the pixels read and processed to take advantage of cached data. We implemented an additional library for LHEASOFT that allows applications to access SLEDs in units of data elements (usually floating point numbers), rather than bytes; the calls are the same, with `ff_` prepended. Tests were performed only on an ext2 file system, and only for file sizes up to 64 MB.

`fimhisto` showed somewhat lower gains than `wc` and `grep`, due to the complexity of the application, but still provided a 15-25% reduction in elapsed time and 30-50% reduction in page faults on files of 48 to 64 MB. Figure 14 shows the familiar pattern of SLEDs offering a benefit above roughly the file system buffer cache size. `fimhisto`'s I/O workload is one fourth writes, which SLEDs does not benefit, and includes data format conversion as well. These factors contribute to the difference in performance gains compared to the above applications.

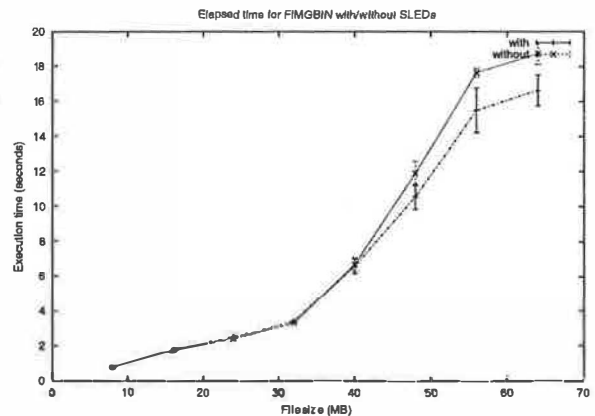


Figure 15: Elapsed time for `fimgbins`, ext2 FS, warm cache, 4x data reduction

We modified `fimgbins` to reorder the reads on its input file according to SLEDs. `fimgbins` rebins an image with a rectangular boxcar filter. The amount of data written is smaller than the input by a fixed factor, typically four or 16. It can also be considered representative of utilities that combine multiple input images to create a single output image. The main `fimgbins` code is in Fortran, so we added Fortran bindings for the principal SLEDs library functions.

Figure 15 shows elapsed time for `fimgbins`. It shows an eleven percent reduction in elapsed time with SLEDs for a data reduction factor of four on file sizes of 48MB or more. This is smaller than the benefit to `fimhisto`, despite similar reductions of 30-50% in pagefaults. We believe this is due to differences in the write path of the array-based code, which is substantially more complex and does more internal buffering, partially defeating our attempts to fully order I/Os. For a data reduction factor of 16, the elapsed time gains were 25-35% over the same range, indicating that the write traffic is an important factor.¹

6 Future Work

The biggest areas of future work are increasing the range of applications that use SLEDs, improving the accuracy of SLEDs both in mechanical details and dynamic system load, and the communication of SLEDs among components of the system (including between file servers and clients). The limitations discussed in section 3.4

¹ A bug in the SLEDs implementation currently limits the rebinning parameters which operate correctly.

need to be addressed.

Devices can be characterized either externally or internally. Hillyer and Sandstå did external device characterization on tape drives, and we have done so on zoned disk drives [Van97]. The devices or subsystems could be engineered to report their own performance characteristics. Cooperation of subsystem vendors will be required to report SLEDs to the file system. Without this data, building true QoS-capable storage systems out of complex components such as HP's AutoRAID [WGSS95] will be difficult, whether done with or without SLEDs.

Work has begun on a migrating hierarchical storage management system for Linux [Sch00]. This will provide an excellent platform for continued development of SLEDs.

7 Conclusion

This paper has shown that Storage Latency Estimation Descriptors, or SLEDs, provide significant improvements by allowing applications to take advantage of the dynamic state of the multiple levels of file system caching. Applications may report expected file retrieval time, prune I/Os to avoid expensive operations, or reorder I/Os to utilize cached data effectively.

Reporting latency, as we have done with `gmc`, is useful for interactive applications to provide the user with more insight into the behavior of the system. This can be expected to improve user satisfaction with system performance, as well as reduce the amount of time users actually spend idle. When users are told how long it will take to retrieve needed data, they can decide whether or not to wait, or productively multitask while waiting.

Pruning I/Os is especially important in heavily loaded systems, and for applications such as `find` that can impose heavy loads. This is useful for network file systems and hierarchical storage management systems, where retrieval times may be high and impact on other users is a significant concern. Because the SLEDs interface is independent of the file system and physical device structure, users do not need to be aware of mount points, volume managers, or HSM organization. Scripts and other utilities built around this concept will remain useful even as storage systems continue to evolve.

Reordering I/Os has been shown, through a series of experiments on `wc` and `grep`, to provide improvement

in execution time of from 50 percent to more than an order of magnitude for file sizes of one to three times the size of the file system buffer cache. Experiments showed an 11-25 percent reduction in elapsed time for `fimhisto` and `fimgbin`, members of a large suite of professional astronomical image processing software. SLEDs-enabled applications have more stable performance in this area as well, showing a gradual decline in performance compared to the sudden step phenomenon at just above the file system buffer cache size exhibited without SLEDs. These experiments were run on ext2, NFS and CD-ROM file systems; the effects are expected to be much more pronounced on hierarchical storage management systems.

Availability and Acknowledgements

The authors thank USC/ISI's Joe Bannister, Ted Faber, John Heidemann, Steve Hotz and Joe Touch for their constructive criticism and support. Quantum's Daniel Stodolsky, as a SLEDs skeptic, and Rick Carlson, as a supporter, both deserve thanks for sharpening the arguments for and against over the last two years. Nokia's Tom Kroeger provided useful feedback on the paper, as well as help with the figures. Finally, our shepherd, Frans Kaashoek, contributed significantly to the structure and clarity of the paper.

Much of the implementation and testing presented here were done by the second author as an intern at Quantum in the Storage Systems Software Unit. Quantum's Hugo Patterson and Tex Schenkkan supported this work.

The source code for SLEDs and the modified applications are available from the first author.

References

- [ADN⁺95] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neeffe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. In *Proc. 15th ACM Symposium on Operating Systems Principles*, pages 109–126. ACM, December 1995.
- [AOG92] David P. Anderson, Yoshitomo Osawa, and Ramesh Govindan. A file system for contin-

- uous media. *Trans. on Computing Systems*, 10(4):311–337, November 1992.
- [BFD97] William J. Bolosky, Robert P. Fitzgerald, and John R. Douceur. Distributed schedule management in the Tiger video fileserver. In *Proc. 16th ACM Symposium on Operating Systems Principles*, pages 212–223. ACM, October 1997.
- [CFKL96] Pei Cao, Edward E. Felten, Anna R. Karlin, and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, November 1996.
- [Dig76] Digital Equipment Corporation. *decsys-tem10 Monitor Calls*, June 1976.
- [FGBA96] Armando Fox, Steven D. Gribble, Eric A. Brewer, and Elan Amir. Adapting to network and client variability via on-demand dynamic distillation. In *Proc. ACM Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 160–170. ACM, October 1996.
- [GA95] James Griffioen and Randy Appleton. Performance measurements of automatic prefetching. In *Proceedings International Conference on Parallel and Distributed Computing Systems*, September 1995.
- [HP89] Robert L. Henderson and Alan Poston. MSS-II and RASH: A mainframe UNIX based mass storage system with a rapid access storage hierarchy file management system. In *Proc. 1989 USENIX Winter Technical Conference*, pages 65–84. USENIX, 1989.
- [HS96a] Bruce Hillyer and Avi Silberschatz. On the modeling and performance characteristics of a serpentine tape drive. In *Proc. ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 170–179. ACM, May 1996.
- [HS96b] Bruce K. Hillyer and Avi Silberschatz. Random I/O scheduling in online tertiary storage systems. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 195–204. ACM, June 1996.
- [Kot94] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74. USENIX Association, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994.
- [Kro00] Thomas M. Kroeger. *Modeling File Access Patterns to Improve Caching Performance*. PhD thesis, University of California Santa Cruz, March 2000.
- [LLJR99] Per Lysne, Gary Lee, Lynn Jones, and Mark Roschke. HPSS at Los Alamos: Experiences and analysis. In *Proc. Sixteenth IEEE Symposium on Mass Storage Systems in cooperation with the Seventh NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 150–157. IEEE, March 1999.
- [LT96] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In *Proc. ACM Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92. ACM, October 1996.
- [MS96] Larry McVoy and Carl Staelin. Lmbench: Portable tools for performance analysis. In *Proceedings of the USENIX 1996 Annual Technical Conference*, pages 279–294. USENIX, January 1996.
- [NAS00] NASA. Lheasoft. Goddard Space Flight Center Laboratory for High Energy Astrophysics, <http://rxte.gsfc.nasa.gov/lheasoft/>, April 2000.
- [NSN⁺97] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Proc. 16th ACM Symposium on Operating Systems Principles*, pages 276–287. ACM, October 1997.
- [PA94] Joseph Pasquale and Eric Anderson. Container shipping: Operating system support for I/O-intensive applications. *IEEE Computer*, 27(3):84–93, March 1994.
- [PGG⁺95] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching.

- In *Proc. 15th ACM Symposium on Operating Systems Principles*, pages 79–95. ACM, December 1995.
- [RW94] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, March 1994.
- [Sat90] Mahadev Satyanarayanan. Scalable, secure, and high available distributed file access. *IEEE Computer*, pages 9–21, May 1990.
- [Sch00] Marc Schaefer. The migration filesystem. web page, April 2000. <http://www-internal.alphanet.ch/~schaefer/mfs.html>.
- [SDH⁺96] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS file system. In *Proc. 1996 USENIX Technical Conference*, pages 1–14. USENIX, January 1996.
- [Shi98] Jamie Shiers. Building a database for the large hadron collider (LHC): the exabyte challenge. In Ben Kobler, editor, *Proc. Sixth NASA Goddard Conference on Mass Storage Systems and Technologies in Cooperation with Fifteenth IEEE Symposium on Mass Storage Systems*, pages 385–396, March 1998.
- [SM99] Olav Sandstål and Roger Midstraum. Low-cost access time model for serpentine tape drive. In *Proc. Sixteenth IEEE Symposium on Mass Storage Systems in cooperation with the Seventh NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 116–127. IEEE, March 1999.
- [Ste97] David C. Steere. Exploiting the non-determinism and asynchrony of set iterators to reduce aggregate file I/O latency. In *Proc. 16th ACM Symposium on Operating Systems Principles*, pages 252–263. ACM, October 1997.
- [Van97] Rodney Van Meter. Observing the effects of multi-zone disks. In *Proc. USENIX '97 Technical Conference*, pages 19–30. USENIX, January 1997.
- [Van98] Rodney Van Meter. SLEDs: Storage latency estimation descriptors. In Ben Kobler, editor, *Proc. Sixth NASA Goddard Conference on Mass Storage Systems and Technologies in Cooperation with Fifteenth IEEE Symposium on Mass Storage Systems*, pages 249–260, March 1998.
- [vI99] Catherine van Ingen. Storage in Windows 2000. presentation, February 1999.
- [WC95] R. W. Watson and R. A. Coyne. The parallel I/O architecture of the high-performance storage system (HPSS). In *Proc. Fourteenth IEEE Symposium on Mass Storage Systems*, pages 27–44. IEEE, September 1995.
- [WGP94] Bruce L. Worthington, Gregory R. Ganger, and Yale N. Patt. Scheduling for modern disk drives and non-random workloads. Technical Report CSE-TR-194-94, University of Michigan, March 1994.
- [WGSS95] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. In *Proc. 15th ACM Symposium on Operating Systems Principles*, pages 96–108. ACM, December 1995.

A Low-Overhead High-Performance Unified Buffer Management Scheme that Exploits Sequential and Looping References*

Jong Min Kim[†]

Jongmoo Choi[†]

Jesung Kim[†]

Sam H. Noh[‡]

Sang Lyul Min[†]

Yookun Cho[†]

Chong Sang Kim[†]

[†]*School of Computer Science and Engineering
Seoul National University
Seoul 151-742, Korea*

[‡]*Department of Computer Engineering
Hong-Ik University
Seoul 121-791, Korea*

Abstract

In traditional file system implementations, the Least Recently Used (LRU) block replacement scheme is widely used to manage the buffer cache due to its simplicity and adaptability. However, the LRU scheme exhibits performance degradations because it does not make use of reference regularities such as sequential and looping references. In this paper, we present a Unified Buffer Management (UBM) scheme that exploits these regularities and yet, is simple to deploy. The UBM scheme automatically detects sequential and looping references and stores the detected blocks in separate partitions of the buffer cache. These partitions are managed by appropriate replacement schemes based on their detected patterns. The allocation problem among the divided partitions is also tackled with the use of the notion of marginal gains. In both trace-driven simulation experiments and experimental studies using an actual implementation in the FreeBSD operating system, the performance gains obtained through the use of this scheme are substantial. The results show that the hit ratios improve by as much as 57.7% (with an average of 29.2%) and the elapsed times are reduced by as much as 67.2% (with an average of 28.7%) compared to the LRU scheme for the workloads we used.

*This work was supported in part by the Ministry of Education under the BK21 program and by the Ministry of Science and Technology under the National Research Laboratory program.

[†]<http://archi.snu.ac.kr/{jmkim,jskim,symin,cskim}>

[‡]<http://ssrnet.snu.ac.kr/~choijm,cho>

[†]<http://www.cs.hongik.ac.kr/~noh>

1 Introduction

Efficient management of the buffer cache by using an effective block replacement scheme is important for improving file system performance when the size of the buffer cache is limited. To this end, various block replacement schemes have been studied [1, 2, 3, 4, 5, 6]. Yet, the Least Recently Used (LRU) block replacement scheme is still widely used due to its simplicity. While simple, it adapts very well to the changes of the workload, and has been shown to be effective when recently referenced blocks are likely to be re-referenced in the near future [7]. A main drawback of the LRU scheme, however, is that it cannot exploit regularities in block accesses such as sequential and looping references and thus, yields degraded performance [3, 8, 9]. In this paper, we present a Unified Buffer Management (UBM) scheme that exploits these regularities and yet, is simple to deploy. The performance gains are shown to be substantial. Trace-driven simulation experiments show that the hit ratios improve by as much as 57.7% (with an average of 29.2%) compared to the LRU scheme for the traces we considered. Experimental studies using an actual implementation of this scheme in the FreeBSD operating system show that the elapsed time is reduced by as much as 67.2% (with an average of 28.7%) compared to the LRU scheme for the applications we used.

1.1 Motivation

The graphs in Figure 1 show the motivation behind this study. First, Figure 1(a) shows the space-time graph of block references from three applications, namely, *cscope*, *c++*, and *postgres* (details of which will be discussed in Section 4), executing concurrently. The

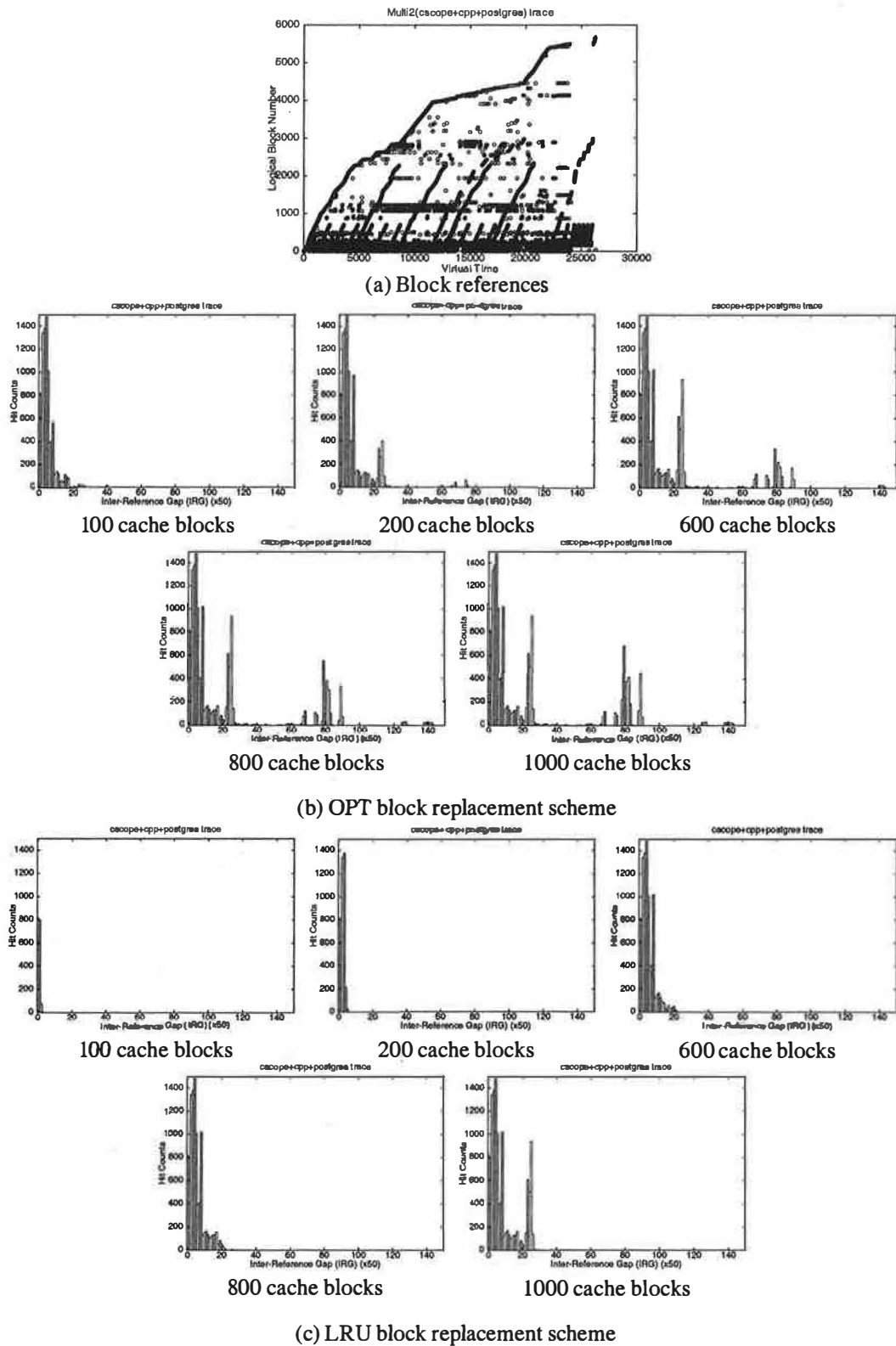


Figure 1: Caching behaviors of the OPT block replacement scheme and the LRU block replacement scheme.

x -axis is the virtual time which ticks at each block reference and the y -axis is the logical block number of the block referenced at the given time. From this graph, we can easily notice sequential and looping reference regularities throughout their execution.

Now consider the graphs in Figures 1(b) and 1(c). They show the Inter-Reference Gap (IRG) distributions of blocks that hit in the buffer cache for the off-line optimal (OPT) block replacement scheme and the LRU block replacement scheme, respectively, as the cache size increases from 100 blocks to 1000 blocks. The x -axis is the IRG and the y -axis is the total hit count with the given IRG.

Observe from the corresponding graphs of the two figures the difference with which the two replacement schemes behave. The main difference comes from how looping references (that is, blocks that are accessed repeatedly with a regular reference interval, which we refer to as the loop period) are treated. The OPT scheme retains the blocks in the increasing order of loop periods as the cache size increases since the scheme chooses a victim block according to the forward distance (i.e., difference between the time of the next reference in the future and the current time). From Figure 1(b), we can see that in the OPT scheme the hit counts of blocks with IRG between 70 and 90 increase gradually as the cache size increases. On the other hand, in the LRU scheme there are no buffer hits at this range of IRGs even when the buffer cache has 1000 blocks. This results from blocks at these IRGs being replaced either by blocks that are sequentially referenced (and thus never re-referenced) or by those with larger IRGs (and thus are replaced before being re-referenced). Although predictable, the regularities of sequential and looping references are not exploited by the LRU scheme, which leads to significantly degraded performance.

From this observation, we devise a new buffer management scheme called the Unified Buffer Management (UBM) scheme. The UBM scheme exploits regularities in reference patterns such as sequential and looping references. Evaluation of the UBM scheme using both trace-driven simulations and an actual implementation in the FreeBSD operating system shows that 1) the UBM scheme is very effective in detecting sequential and looping references, 2) the UBM scheme manages sequentially-referenced and looping-referenced blocks similarly to the OPT scheme, and 3) the UBM scheme shows substantial performance improvements.

1.2 The Remainder of the Paper

The remainder of this paper is organized as follows. In the next section, we review related work. In Section 3, we explain the UBM scheme in detail. In Section 4, we describe our experimental environments and compare the performance of the UBM scheme with those of previous schemes through trace-driven simulations. In Section 5, an implementation of the UBM scheme in the FreeBSD operating system is evaluated. Finally, we provide conclusions and directions for future research in Section 6.

2 Related Work

In this section, we place previous page/block replacement schemes into the following three groups and in turn, survey the schemes in each group.

- Replacement schemes based on frequency and/or recency factors.
- Replacement schemes based on user-level hints.
- Replacement schemes making use of regularities of references such as sequential references and looping references.

The FBR (Frequency-based Replacement) scheme by Robinson and Devarakonda [1], the LRU-K scheme by O'Neil *et al.* [2], the IRG (Inter-Reference Gap) scheme by Phalke and Gopinath [5], and the LRFU (Least Recently/Frequently Used) scheme by Lee *et al.* [6] fall into the first group. The FBR scheme chooses a victim block to be replaced based on the frequency factor differing from the Least Frequently Used (LFU) mainly in that it considers correlations among references. The LRU-K scheme bases its replacement decision on the blocks' k th-to-last reference, while the IRG scheme's decision is based on the inter-reference gap factor. The LRFU scheme considers both the recency and frequency factors of blocks. These schemes, however, show limited performance improvements because they do not consider regular references such as sequential and looping references.

Application-controlled file caching by Cao *et al.* [4] and informed prefetching and caching by Patterson *et al.* [10] are schemes based on user-level hints. These schemes choose a victim block to be replaced based

on user-provided hints on application reference characteristics, allowing different replacement policies to be applied to different applications. However, to obtain user-level hints, users need to accurately understand the characteristics of block reference patterns of applications. This requires considerable effort from users limiting the applicability.

The third group of schemes considers regularities of references, and the 2Q scheme by Johnson and Shasha [3], the SEQ scheme by Glass and Cao [8], and the EELRU (Early Eviction LRU) scheme by Smaragdakis *et al.* [9] fall into this group. The 2Q scheme quickly removes from the buffer cache sequentially-referenced blocks and looping-referenced blocks with long loop periods. This is done by using a special buffer called the *A1in* queue in which all missed blocks are initially placed and from which the blocks are replaced in the FIFO order after short residence. On the other hand, the scheme holds looping-referenced blocks with short loop periods in the main buffer cache by using a ghost buffer called the *A1out* queue in which the addresses of blocks replaced from the *A1in* queue are temporarily placed to discriminate between frequently referenced blocks and infrequently referenced ones. When a block is re-referenced while its address is in the *A1out* queue, it is promoted to the main buffer cache. The 2Q scheme, however, has two drawbacks. One is that an additional miss has to occur for a block to be promoted to the main buffer cache from the *A1out* queue. The other is that a careful tuning is required for two control parameters, that is, the size of the *A1in* queue and the size of the *A1out* queue, which may be sensitive to the types of workload.

The SEQ scheme detects long sequences of page faults and applies the Most Recently Used (MRU) scheme to those pages. However, in determining the victim page, it does not distinguish sequential and looping references. The EELRU scheme confirms the existence of looping references by examining aggregate recency distributions of referenced pages and changes the page eviction points using a simple on-line cost/benefit analysis. The EELRU scheme, however, does not distinguish between looping references with different loop periods.

3 The Unified Buffer Management Scheme

The Unified Buffer Management (UBM) scheme is composed of the following three main modules.

Detection This module automatically detects sequential and looping references. After the detection, block references are classified into sequential, looping, or other references.

Replacement This module applies different replacement schemes to the blocks belonging to the three reference patterns according to the properties of each pattern.

Allocation This module allocates the limited buffer cache space among the three partitions corresponding to sequential, looping, and other references.

In the following subsections, we give a detailed explanation of each of these modules.

3.1 Detection of Sequential and Looping References

The UBM scheme automatically detects sequential, looping, and other references according to the following rules:

Sequential references that are consecutive block references occurring only once.

Looping references that are sequential references occurring repeatedly with a regular interval.

Other references that are detected neither as sequential nor as looping references.

Figure 2 shows the classification process of the UBM scheme. Note that looping references are initially detected as sequential until they are re-referenced.

For on-line detection of sequential and looping references, information about references to blocks in each file is maintained in an abstract form. The elements needed are shown in Figure 3.

Information for each file is maintained as a 4-tuple consisting of a file descriptor (*fileID*), a start block number (*start*), an end block number (*end*), and a loop period (*period*). A reference is categorized as a sequential reference after a given number of consecutive references are made. For a sequential reference the loop period is ∞ , while for a looping reference its value is the actual loop period. In real systems, the loop period fluctuates by various factors including the degree

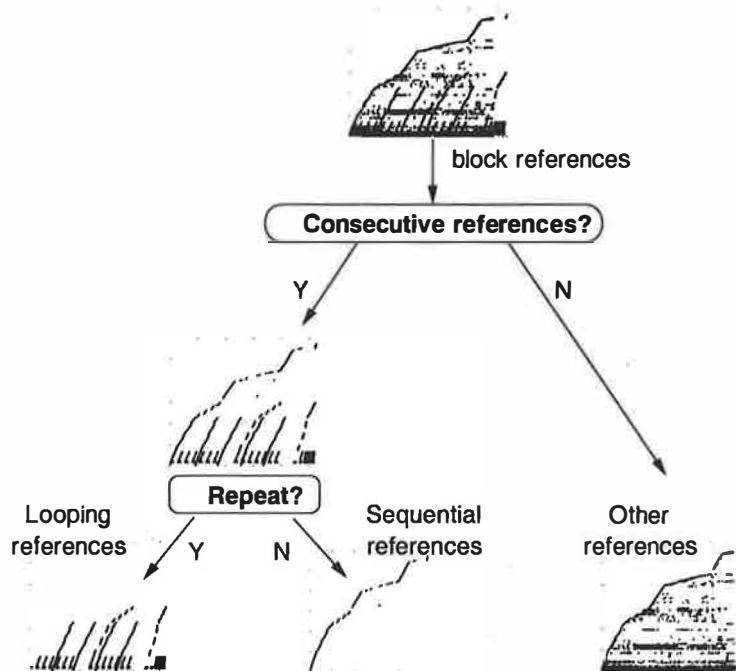


Figure 2: Classification process of the UBM scheme.

of multiprogramming and scheduling. Hence, this value is set as an exponential average of measured loop periods. Also, since a looping reference is initially detected as a sequential reference, its blocks are managed just like those belonging to a sequential reference until they are re-referenced. This may make them miss the first time they are re-referenced if there is not enough space in the cache (cf. Section 4.7).

The resulting table keeps information for sequences of consecutive block references that are detected up to the current time and is updated whenever a block reference occurs. In most UNIX file systems, sequences of consecutive block references are detected by using vnode numbers and consecutive block addresses.

Figure 4 shows an example of sequential and looping references, and the data structure that is used to maintain information for these references. In the figure, the file with fileID 3 is a sequential reference as it has ∞ as its loop period. Files with fileID 1 and 2 are looping references with loop periods of 80 and 40, respectively.

3.2 Block Replacement Schemes

The detection mechanism results in files being categorized into three types, that is, sequential, looping, and other references. The buffer cache is divided into three

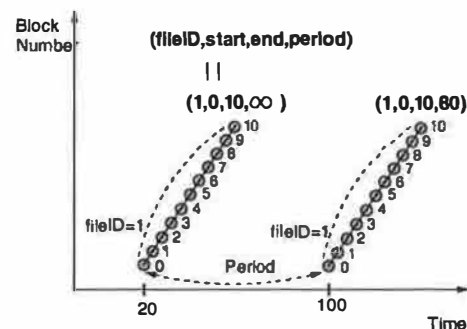


Figure 3: Elements for representing sequential and looping references.

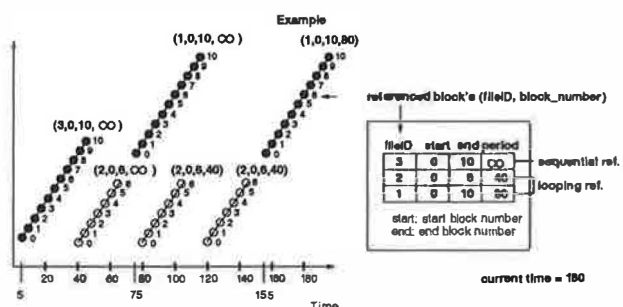


Figure 4: Example of detection: sequential and looping references.

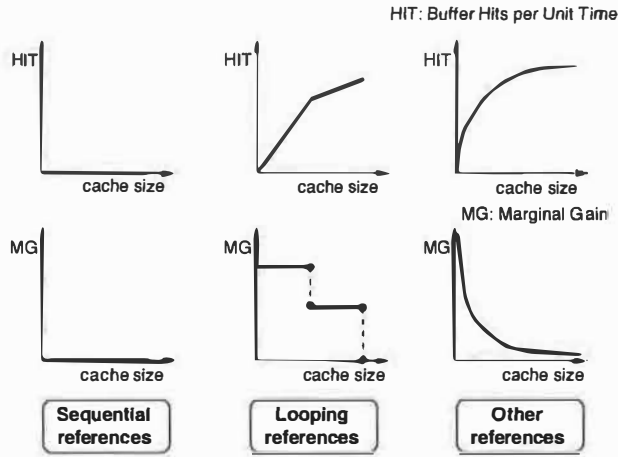


Figure 5: Typical curves of buffer hits per unit time and marginal gain values.

partitions to accommodate the three different types of references. Management of the partitions must be done according to the reference characteristics of blocks belonging to each partition.

For the partition that holds sequential references, it is a simple matter. Sequentially-referenced blocks are never re-referenced. Hence, the referenced blocks need not be retained and therefore, the MRU replacement policy is used.

For the partition that holds looping references, victim blocks to be replaced are chosen based on their loop periods because their re-reference probabilities depend on these periods. To do so, we use a period-based replacement scheme that replaces blocks in decreasing order of loop periods, and the MRU block replacement scheme among blocks with the same loop period.

Finally, within the partition that holds other references, victim blocks to be replaced can be chosen based on their recency, frequency, or a combination of the two factors. Hence, we may use any of previously proposed replacement schemes including the Least Recently Used (LRU), the Least Frequently Used (LFU), the LRU-K, and the Least Recently/Frequently Used (LRFU) as long as they have a model that approximates the hit ratio for a given buffer size to compute the marginal gain, which will be explained in the next subsection. In this paper, we assume the LRU replacement scheme.

3.3 Buffer Allocation Based on Marginal Gain

The buffer cache has now been divided into three partitions that are being managed separately. An important

problem that should be addressed then is how to allocate the blocks in the cache among the three partitions. To this end, we use the notion of marginal gains, which has frequently been used in resource allocation strategies in various computer systems areas [10, 11, 12, 13].

Marginal gain is defined as $MG(n) \approx Hit(n) - Hit(n-1)$, which specifies the expected number of extra buffer hits per unit time that would be obtained by increasing the number of allocated buffers from $(n-1)$ to n , where $Hit(n)$ is the expected number of buffer hits per unit time using n buffers. In the following, we explain how to predict the marginal gains of sequential, looping, and other references, as the buffer cache is partitioned to accommodate each of these types of references.

The expected number of buffer hits per unit time of sequential references when using n buffers is $Hit_{seq}(n) = 0$, and thus, the expected marginal gain is always $MG_{seq}(n) = 0$.

For looping references, the expected number of buffer hits per unit time and the expected marginal gain value are calculated as follows.

First, for a looping reference, $loop_i$, with loop length l_i and loop period p_i , the expected number of buffer hits per unit time when using n buffers is $Hit_{loop_i}(n) = \min[l_i, n]/p_i$. Thus, if $n \leq l_i$, the expected marginal gain is $MG_{loop_i}(n) = n/p_i - (n-1)/p_i = 1/p_i$ and if $n > l_i$, $MG_{loop_i}(n) = l_i/p_i - l_i/p_i = 0$.

Now, assume that there are L looping references $\{loop_1, \dots, loop_i, \dots, loop_L\}$, where the loops here are arranged in the increasing order of loop periods. Let i_{max} be the maximum of i such that $m = \sum_{k=1}^i l_k < n$, where n is the number of buffers in the partition for looping references. If $i_{max} = L$, then all loops can be held in the buffer cache, and hence $Hit_{loop}(n) = \sum_{k=1}^L l_k/p_k$, and $MG_{loop}(n) = 0$. Consider now, the more complicated case where $i_{max} < L$. The expected number of buffer hits per unit time of these looping references when using n buffers is $Hit_{loop}(n) = \sum_{k=1}^{i_{max}} l_k/p_k + \min[l_{i_{max}+1}, n - \sum_{k=1}^{i_{max}} l_k]/p_{i_{max}+1}$. (Recall that we are using the period-based replacement scheme to manage the partition for looping references. Hence, there can be no loops within this partition that has loop period longer than $p_{i_{max}+1}$.) Hence, the expected marginal gain is $MG_{loop}(n) = 1/p_{i_{max}+1}$.

Finally, for the partition that holds the other references and which is managed by the LRU replacement scheme, the expected number of buffer hits per unit time and the expected marginal gain value can be calculated from the

buffer hit ratio using ghost buffers [11, 10] and/or the Belady's lifetime function [14]. Ghost buffers, sometimes called *dataless* buffers, are used to estimate the number of buffer hits per unit time for cache sizes larger than the current size when the cache is managed by the LRU replacement scheme. A ghost buffer does not contain any data blocks but maintains control information needed to count cache hits. The prediction of the buffer hit ratio using only ghost buffers is impractical due to the overhead of measuring the hit counts of all LRU stack positions individually. In the UBM scheme, we use an approximation method suggested by Choi *et al.* [13]. The proposed method utilizes the Belady's lifetime function, which is well-known to approximate the buffer hit ratio of references that follow the LRU model. Specifically, the hit ratio with n buffers is given by Belady's lifetime function as

$$hit_{other}(n) = h_1 + h_2 + h_3 + \dots + h_n \approx 1 - c * n^{-k}$$

where c and k are control parameters. Specific c and k values can be calculated on-line by using measured buffer hit ratios at pre-set cache sizes. Ghost buffers are used to determine the hit ratios at these pre-set cache sizes. The overhead of using ghost buffers in this case is minimal as accurate LRU stack positions of referenced blocks need not be located. For example, to calculate the values of the control parameters, c and k , buffer hit ratios at a minimum of two cache sizes, say, p and q , where $p \neq q$ are required. Using these values and the equation of the lifetime function, we can calculate the values of c and k . Then, the expected number of buffer hits per unit time is given by $Hit_{other}(n) = hit_{other}(n) \times \frac{n_{other}}{n_{total}}$ where n_{other} and n_{total} are the number of other references and the number of total references, respectively, during the observed period. Finally, the expected marginal gain is simply $MG_{other}(n) = Hit_{other}(n) - Hit_{other}(n-1)$.

Figure 5 shows typical curves of both buffer hits per unit time and marginal gain values of sequential, looping, and other references as the number of allocated buffers increases. In the UBM scheme, since the marginal gain of sequential references, $MG_{seq}(n)$, is always zero, the buffer manager does not allocate more than one buffer to the corresponding partition, except when buffers are not fully utilized. That is, only when there are free buffers at the initial stage of allocation, more than one buffer may be allocated to this partition. Thus, in general, buffer allocation is determined between the partitions that hold the looping-referenced blocks and the other-referenced blocks. The UBM scheme tries to maximize the expected number of total buffer hits by dynamically controlling the allocation so that the marginal gain value of looping references, $MG_{loop}(n)$, and the marginal gain value of other ref-

erences, $MG_{other}(C - n)$, where C is the cache size, converge to the same value.

3.4 Interaction Among the Three Modules

Figure 6 shows the overall interactions between the three modules of the UBM scheme. Whenever a block reference occurs in the buffer cache, the detection module updates and/or classifies the reference into one of the sequential, looping, or other reference types (step (1) in Figure 6). In this example, assume that a miss has occurred and the reference is classified as an other reference. As a miss has occurred the buffer allocation module is called to get additional buffer space for the referenced block (step (2)). The buffer allocation module would normally compare the marginal gain values of looping and other references and choose the one with a smaller marginal gain value and send a replacement request to the corresponding cache partition as shown in step (3). However, if there is space allocated to a sequential reference, such space is always deallocated first. The cache management module of the selected cache partition decides a victim block to be replaced using its replacement scheme (step (4)) and deallocates the buffer space of the victim block to the allocation module (step (5)). The allocation module forwards this space to the cache partition for other-referenced blocks (step (6)). Finally, the referenced block is fetched from disk into the buffer space (step (7)).

4 Performance Evaluation

In this section and the next, we discuss the performance of the UBM scheme. This section concentrates on the simulation study, while the next section focuses on the implementation study.

In this section, the performance of the UBM scheme is compared with those of the LRU, 2Q, SEQ, EELRU, and OPT schemes through trace-driven simulations¹. We also compare the performance of the UBM scheme with that of application-controlled file caching through trace-driven simulations with the same multiple application trace used in [4]. We did not compare the performance of the UBM and those of schemes based on recency and/or frequency factors such as FBR, LRU-K,

¹Though the SEQ and EELRU schemes were originally proposed as page replacement schemes, they can also be used as block replacement schemes.

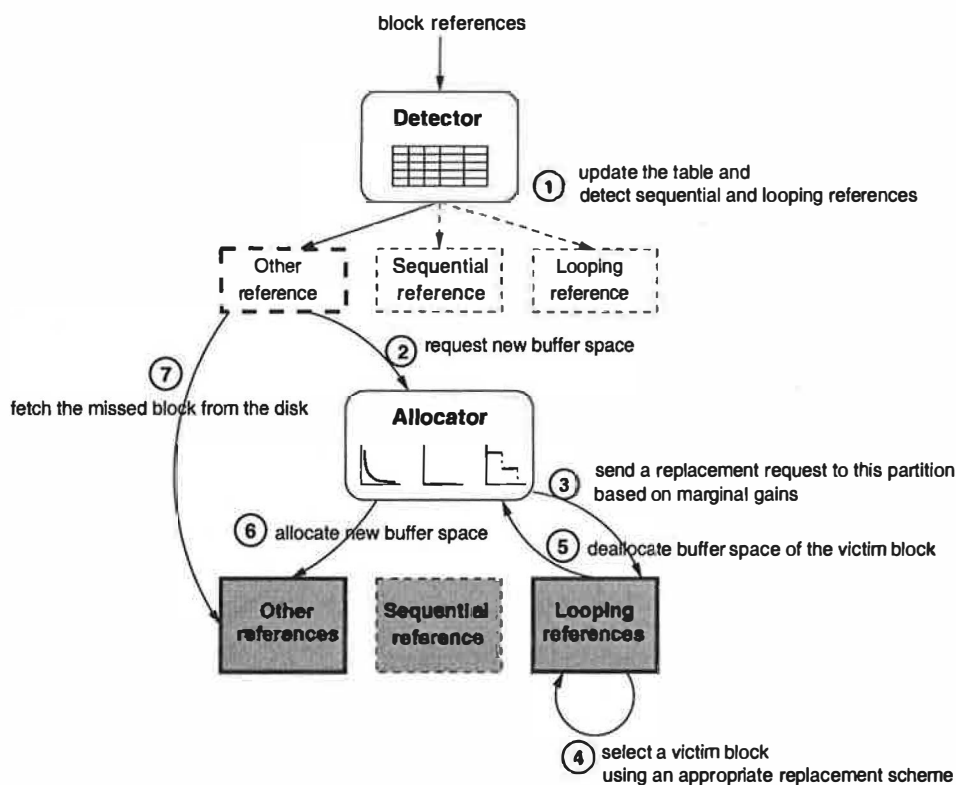


Figure 6: Overall structures of the UBM scheme.

Table 1: Characteristics of the traces used.

Trace	Applications executed concurrently	# of references	# of unique blocks
Multi1	cscope, cpp	15858	2606
Multi2	cscope, cpp, postgres	26311	5684
Multi3	cpp, gnuplot, glimpse, postgres	30241	7453

and LRFU since the benefits from the two factors are largely orthogonal and any of the latter schemes can be used to manage other references in the UBM scheme. We first describe the experimental setup and then present the performance results.

4.1 Experimental Setup

Traces used in our simulations were obtained by concurrently executing diverse applications on the FreeBSD operating system running on an Intel Pentium PC. The characteristics of the applications are described below.

cpp Cpp is the GNU C compiler pre-processor. The total size of C sources used as input is roughly

11MB. During execution, observed block references are sequential and other references.

cscope Cscope is an interactive C source examination tool. The total size of C sources used as input is roughly 9MB. It exhibits looping references with an identical loop period and other references.

glimpse Glimpse is a text information retrieval utility. The total size of text files used as input is roughly 50MB. Its block reference characteristic is diverse - it shows sequential references, looping references with different loop periods, and other references.

gnuplot Gnuplot is an interactive plotting program. The size of raw data used as input is 8MB. Looping references with an identical loop period and

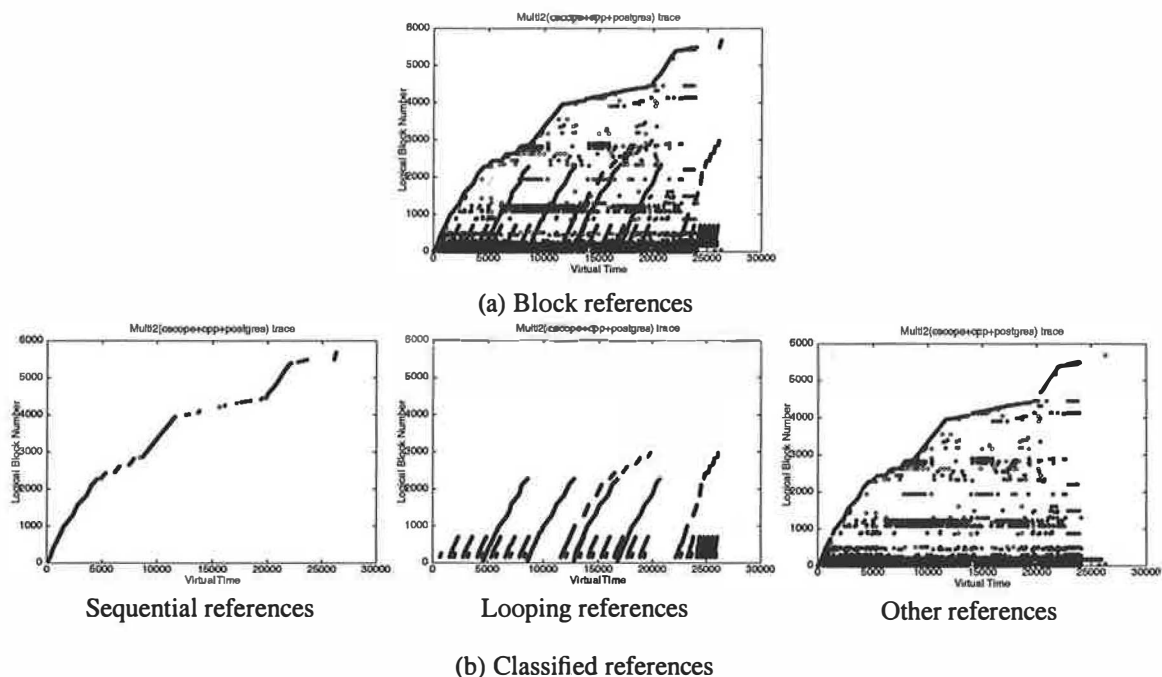


Figure 7: Reference classification results (Trace: *Multi2*).

other references were observed during execution.

postgres Postgres is a relational database system from the University of California at Berkeley. We used join queries among four relations, namely, *twot-houstup*, *twentythoustup*, *hundredthoustup*, and *twohundredthoustup*, which were made from a scaled-up Wisconsin benchmark. The sizes of each relation are approximately 150KB, 1.5MB, 7.5MB, and 15MB. It exhibits sequential references, looping references with different loop periods, and other references.

mpeg_play Mpeg_play is an MPEG player from the University of California at Berkeley. The size of the MPEG video file used as input is 5MB. Sequential references dominate in this application.

We used three multiple application traces in our experiments. They are denoted by *Multi1*, *Multi2*, and *Multi3*, and their characteristics are shown in Table 1.

We built simulators for the LRU, 2Q, SEQ, EELRU, and OPT schemes as well as the UBM scheme. Unlike the UBM scheme that does not have any parameters that need to be tuned, the 2Q, SEQ, and EELRU schemes have one or more parameters whose settings may affect the performance. For example, in the 2Q scheme the parameters are the sizes of the *A1in* and *A1out* queues.

The parameters of the SEQ scheme are threshold values used to choose victim blocks among consecutively missed blocks. Finally, for the EELRU scheme, the early and late eviction points from which a block is replaced, have to be set. In our experiments, we used the values suggested by the authors of each of the schemes.

4.2 Detection Results

Figure 7 shows the classifications resulting from the detection module for the *Multi2* trace. The *x*-axis is the virtual time and the *y*-axis is the logical block number. The figures are space-time graphs with Figure 7(a) showing block references for the whole trace and Figure 7(b) showing how the detection module classified the sequential, looping, and other references from the original references. The results indicate that the UBM scheme accurately classifies the three reference patterns.

4.3 IRG Distribution Comparison of the UBM Scheme with the OPT Scheme

Figure 8 shows the caching behaviors of the OPT and UBM schemes for the *Multi2* trace. Compare these results with those shown in Figures 1(b) and 1(c), which use the same trace. Recall that these graphs show the

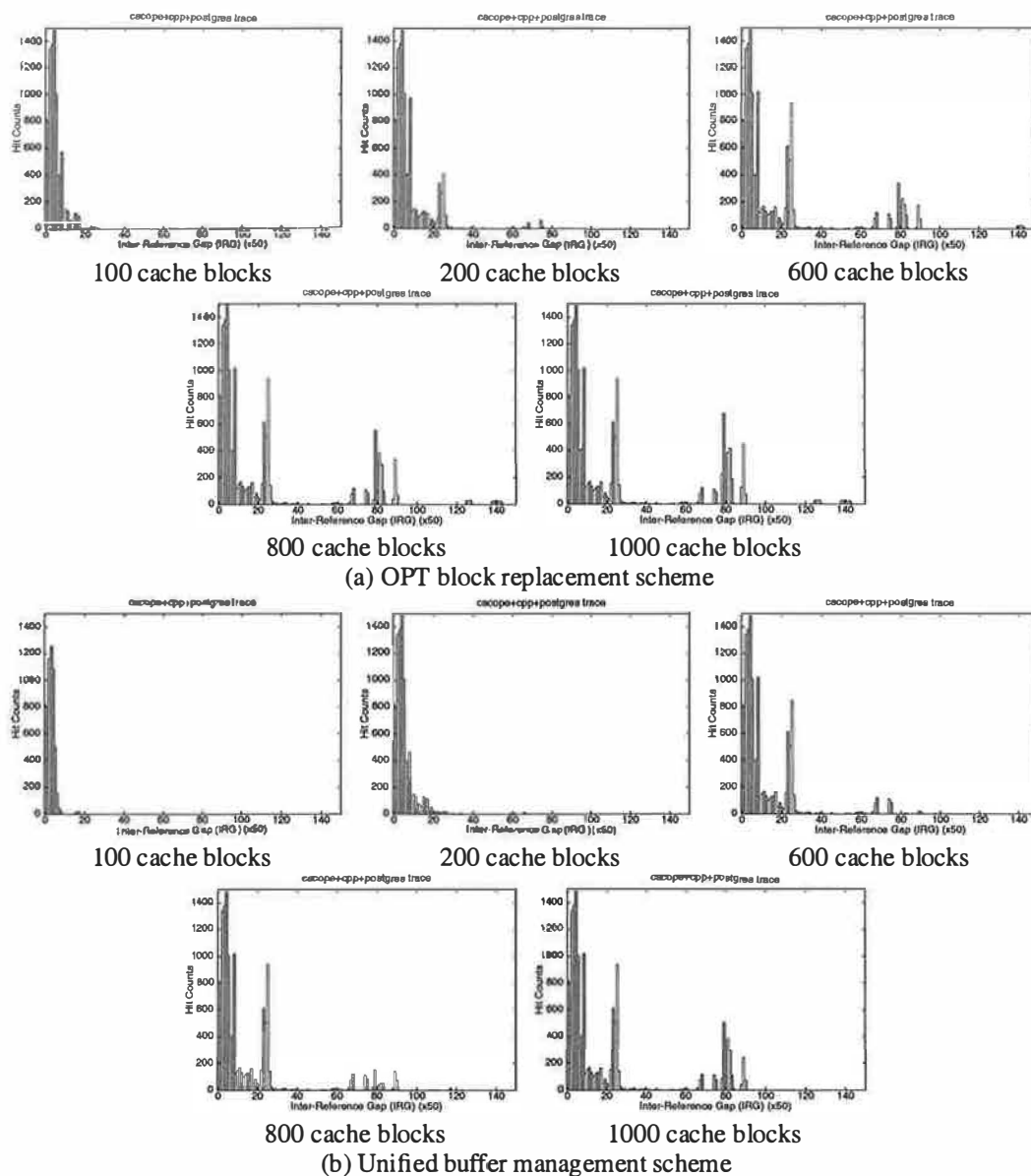


Figure 8: Caching behaviors of the OPT and UBM schemes.

hit counts in the buffer cache using IRG distributions of referenced blocks. We note that the UBM scheme is very closely mimicking the behavior of the OPT scheme.

4.4 Performance Comparison of the UBM Scheme with Other Schemes

Figure 9 shows the buffer hit ratios of the UBM scheme and other replacement schemes as a function of cache size with the block size set to 8KB. For most cases, the UBM scheme shows the best performance. Further analysis for each of the schemes is as follows.

The SEQ scheme shows fairly stable performance for all cache sizes. The reason behind its good performance is that it quickly replaces sequentially-referenced blocks that miss in the buffer cache. However, since the scheme does not consider looping references, it shows worse performance than the UBM scheme.

The 2Q scheme shows better performance than the LRU scheme for most cases because it quickly replaces sequentially-referenced blocks and looping-referenced blocks with long loop periods. However, when the cache size is large (caches with 1800 or more blocks for the *Multi1* trace, 2800 or more blocks for the *Multi2* trace, and 3600 or more blocks for the *Multi3* trace), the scheme shows worse performance than the LRU

scheme. There are two reasons behind this. First, since the scheme replaces all newly referenced blocks after holding it in the buffer cache for a short time, whenever these blocks are re-referenced, additional misses occur. The ratio of these additional misses to total misses increases as the cache size increases resulting in a significant impact on the performance of the buffer cache. Second, the scheme does not distinguish between looping-referenced blocks with different loop periods. The performance of the 2Q scheme does not gradually increase with the cache size, but rather surges beyond some cache size and then holds steady. Also the 2Q scheme exhibits rather anomalous behavior for the *Multi2* and *Multi3* traces. When the cache sizes are about 2200 blocks and 2800 blocks for *Multi2* and *Multi3*, respectively, the buffer hit ratio of the scheme decreases as the cache size increases. A careful inspection of results reveals that when the cache size increases the *Alout* queue size increases accordingly and this results in a situation where blocks that should not be promoted are promoted to the main buffer cache leading to such an anomaly.

The EELRU scheme shows similar or better performance than the LRU scheme as the cache size increases. However, since the scheme chooses a victim block to be replaced based on aggregate recency distributions of referenced blocks, it does not replace quickly enough the sequentially-referenced blocks and looping-referenced blocks that have long loop periods. Further, like the 2Q scheme, it does not distinguish between looping-referenced blocks with different loop periods. Hence, it does not fair well compared to the UBM scheme.

The LRU scheme shows the worst performance for most cases because it does not give any consideration to the regularities of sequential and looping references.

Finally, the UBM scheme replaces sequentially-referenced blocks quickly and holds the looping-referenced blocks in increasing order of loop periods based on the notion of marginal gains as the cache size increases. Consequently, the UBM scheme improves the buffer hit ratios by up to 57.7% (for the *Multi1* trace with 1400 blocks) compared to the LRU scheme with an average increase of 29.2%.

4.5 Results of Dynamic Buffer Allocation

Figure 10(a) shows the distribution of the buffers allocated to the partitions that hold sequential, looping, and other references as a function of (virtual) time when the buffer cache size is 1000 blocks. Until time 2000, the

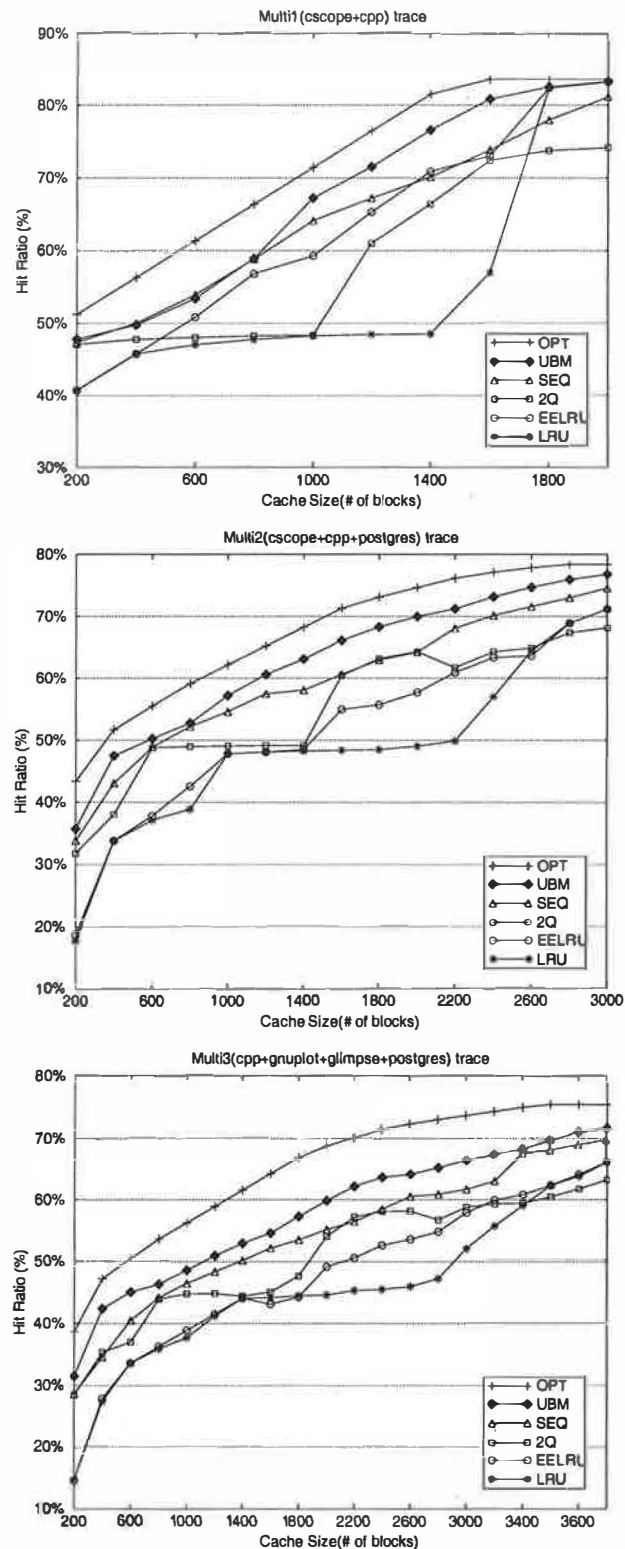


Figure 9: Performance comparison of the UBM scheme with other schemes.

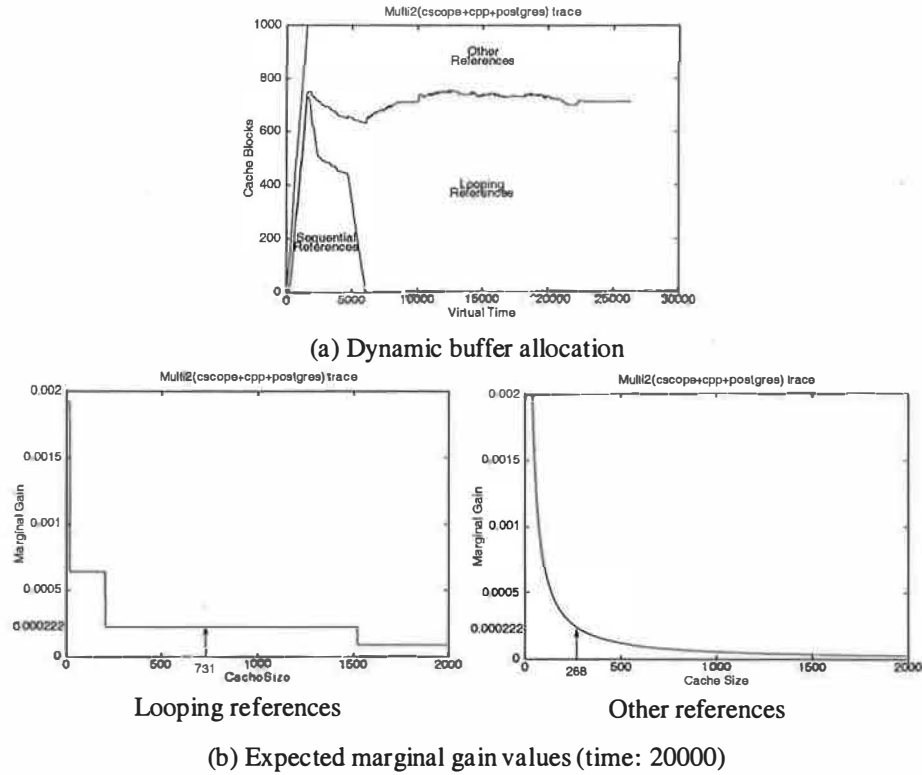


Figure 10: Results of dynamic buffer allocation (cache size: 1000 blocks, trace: *Multi2*).

buffer cache is not fully utilized. Hence, buffers are allocated to any reference that requests it, resulting in the partition that holds sequentially-referenced blocks being allocated a considerable number of blocks. After time 2000, all the buffers have been consumed, and hence, the number of buffers allocated to the partition for sequentially-referenced blocks decreases, while allocations to the partitions for looping and other references start to increase. As a result, at around time 6000, only one buffer is allocated to the partition for sequentially-referenced blocks. From about time 10000, the allocations to the partitions for the looping and other references converge to a steady-state value.

Figure 10(b) shows marginal gains as a function of allocated buffers of looping and other references that are calculated at time 20000 of Figure 10(a). Since there are several looping references with different loop periods in the *Multi2* trace, from the left figure, we can see that the expected marginal gain values of the looping references, $MG_{loop}(n)$, decrease step-wise as the number of allocated buffers, n , increases. The figure on the right shows the expected marginal gain values of the other references, $MG_{other}(n)$, that decrease gradually. The UBM scheme dynamically controls the number of allocated buffers to each partition so that the marginal

gain values of the two partitions converge to the same value. At time 20000, the two marginal gain values converge to 0.000222 and the number of allocated buffers to the partitions for the looping and other references is 731 blocks and 268 blocks, respectively.

4.6 Performance Comparison of the UBM Scheme with Application-controlled File Caching

The application-controlled file caching (ACFC) scheme [4] is a user-hint based approach to buffer cache management, which is in contrast to the UBM scheme that requires no such hints. To compare the performance of these two schemes, we used the ULTRIX multiple application (*postgres* + *cscope* + *linking the kernel*) trace in [4].

Figure 11 shows the buffer hit ratios of the UBM scheme, two ACFC schemes, namely, *ACFC(HEURISTIC)* and *ACFC(RMIN)*, and the LRU, 2Q, SEQ, EELRU, and OPT schemes when the cache size increases from 4M to 16M. The *ACFC(HEURISTIC)* scheme uses user-level hints for each application, while the *ACFC(RMIN)* scheme uses

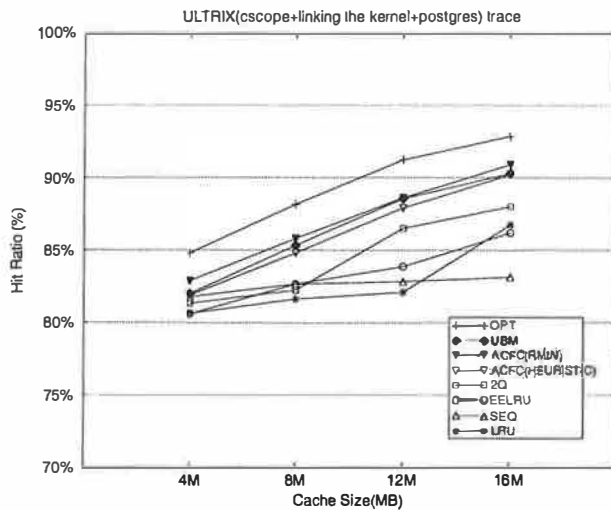


Figure 11: Performance comparison of the UBM scheme with the application-controlled file caching scheme.

the optimal off-line strategy for each application. The results for the two ACFC schemes were borrowed from [4] while the results for all the other schemes were obtained from simulations. The results show that the hit ratios of the UBM scheme, which does not make use of any external hints, are comparable to those of the ACFC(RMIN) scheme and higher than those of the ACFC(HEURISTIC) scheme.

4.7 Warm/Cold Caching Effects

All experiments so far were done with cold caches. To evaluate the performance of the UBM scheme at steady-state, we performed additional simulations with warmed-up caches. In the experiments, we run initially a long-run workload (*sdet_benchmark*²) through the buffer cache and after the cache was warmed up, we run both the long-run workload and the target workload (*cscope* + *cpp* + *postgres*) concurrently. The cache statistics were collected only after the cache was warmed up.

Experimental results that show the warm/cold caching effects of the UBM scheme are given in Figure 12 and Table 2. The graphs of Figure 12 show that when the cache size is small, the performance improvements by the UBM scheme with warmed-up caches over the LRU scheme are similar to those with cold caches. As the cache size increases, however, the performance increase by the UBM scheme with warmed-up caches is reduced

²Sdet is the SPEC SDET benchmark that simulates a multiprogramming environment.

when compared with the performance increase with cold caches since sequential references are not allowed to be cached at all. Specifically, when the cache is warmed up, blocks belonging to sequential references are not allowed to be cached. If those blocks are re-referenced with a regular interval in the future (i.e., if they turn into looping references), the UBM scheme has to reread them from the disks. In cold caches, however, many of them are reread from the cache partition that holds sequentially-referenced blocks because when the cache is cold, more than one block can be allocated to the partition for the sequentially referenced blocks.

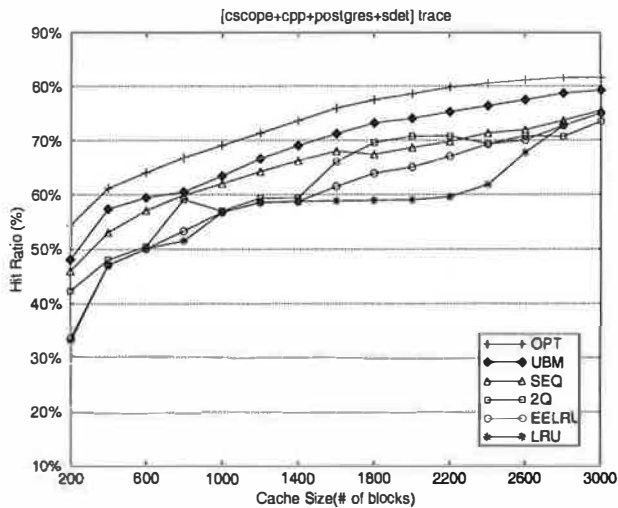
The resulting performance degradation, however, is not significant as we can see from Table 2 that summarizes the performance improvements by the UBM scheme for both cold caches and warmed-up caches - the difference in the average performance improvement between cold caches and warmed-up caches is less than 1%. Although the overall performance degradation is negligible, the additional misses may adversely affect the performance perceived by the user due to increased start-up time. As future work, we plan to explore an allocation scheme where blocks are allocated even for sequential references based on the probability that a sequential reference will turn into a looping reference.

5 Implementation of UBM in the FreeBSD Operating System

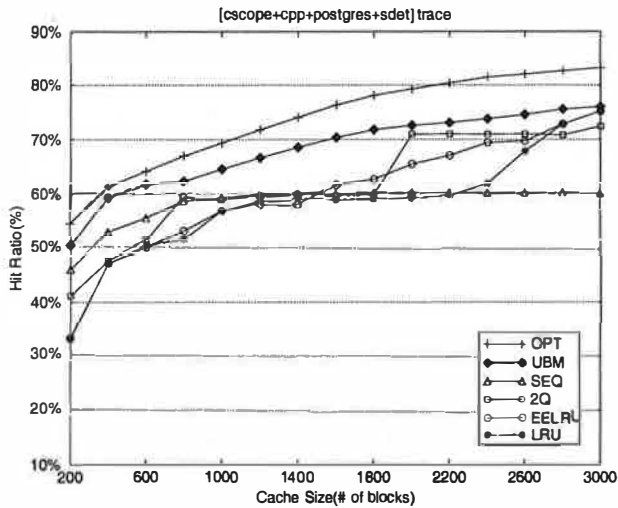
The UBM scheme was integrated into the FreeBSD operating system running on a 133MHz Intel Pentium PC with 128MB RAM and a 1.6GB Quantum Fireball hard disk. For the experiments, we used five real applications (*cpp*, *cscope*, *glimpse*, *postgres*, and *mpeg_play*) that were explained in Section 4. We ran several combinations of three or more of these applications concurrently and measured the elapsed time of each application under the UBM and SEQ schemes as well as under the built-in LRU scheme when the cache sizes are 8MB, 12MB, and 16MB with the block size set to 8KB.

5.1 Performance Measurements

Figure 13 shows the elapsed time of each individual application under the UBM, SEQ and LRU schemes. As expected, the UBM scheme shows better performance than the LRU scheme. In the figure, since the *postgres* and *cscope* applications access large files repeatedly with a regular interval, they show better improvement



(a) With cold caches



(b) With warmed-up caches

Figure 12: The cold/warm caching effects of the UBM scheme in trace-driven simulations (Trace: *cscope* + *cpp* + *postgres* + *sdet*).

in the elapsed time than other applications. On the other hand, the *mpeg_play* application does not show as much improvement because it accesses a large video file sequentially.

Overall, the UBM scheme reduces the elapsed time by up to 67.2% (the elapsed time of the *postgres* application for the *cpp* + *postgres* + *cscope* + *mpeg_play* case with 16MB cache size) compared to the LRU scheme with an average improvement of 28.7%. We note that improvements by the UBM scheme on the elapsed time are comparable to those on the buffer hit ratios we observed in the previous section.

There are two main benefits from using the UBM scheme. The first is from detecting looping references,

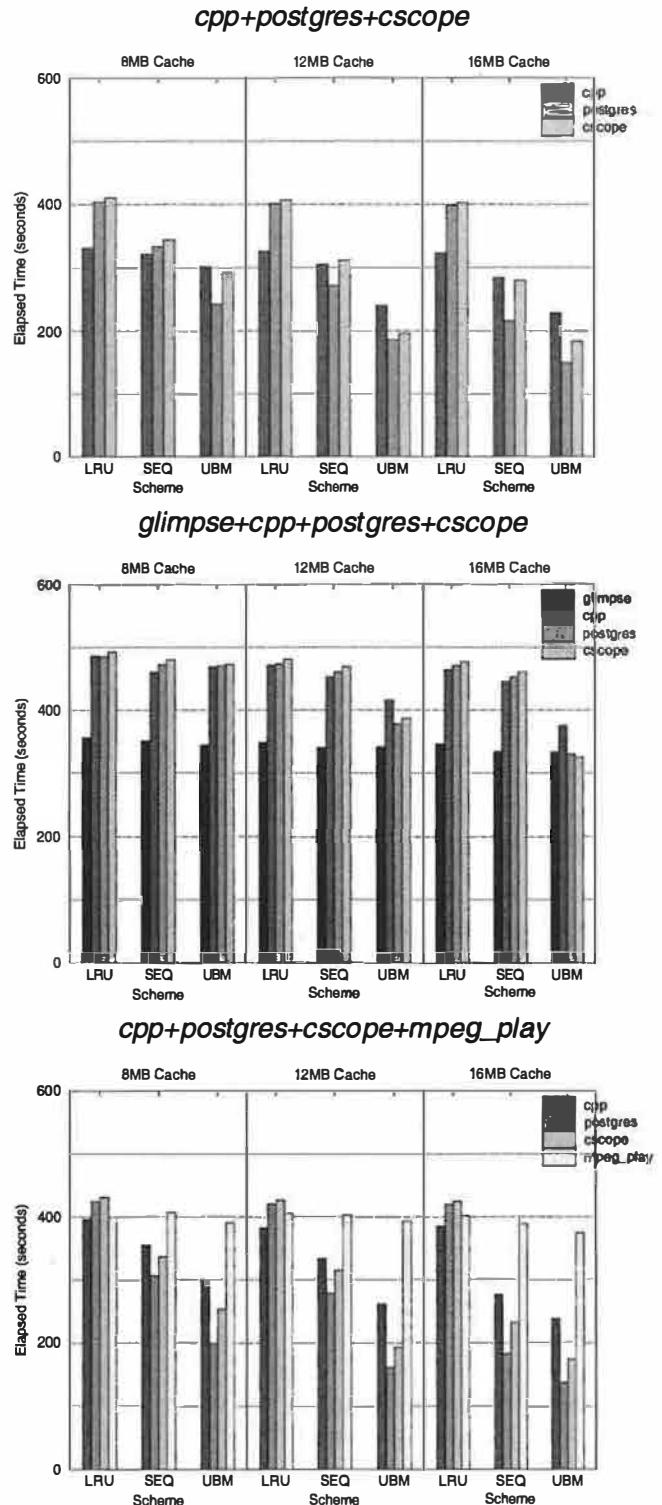


Figure 13: Performance of the UBM scheme integrated into the FreeBSD.

Table 2: Comparison of performance improvements of the UBM scheme compared to the LRU scheme (Trace: *cscope + cpp + postgres + sdet*).

Improvements with cold caches	Improvements with warmed-up caches
avg. 19.6% (max. 26.2%)	avg. 18.9% (max. 22.6%)

managing them by a period-based replacement policy, and allocating buffer space to them based on marginal gains. The second is from giving preference to blocks belonging to sequential references when a replacement is needed. To quantify these benefits, we compared the UBM scheme with the SEQ scheme. The results in Figure 13 show that there is still a substantial difference in the elapsed time between the UBM scheme and the SEQ scheme indicating that the benefit from carefully handling looping references is significant.

5.2 Effects of Run-Time Overhead of the UBM Scheme

To measure the run-time overhead of the UBM scheme, we executed a CPU-bound application (*cpu_bound*) that executes an *ADD* instruction repeatedly, along with the other applications.

Figure 14 shows the run-time overhead of the UBM scheme for two combinations of applications, namely, *cpu_bound+cpp+postgres+cscope* and *cpu_bound+glimpse+cpp+postgres+cscope* when the cache size is 12MB. The elapsed time of the *cpu_bound* application increases slightly by around 5% when using the UBM scheme compared with that when using the LRU scheme. A major source of this overhead comes from the operations to manipulate the ghost buffers and to calculate the marginal gain values. Currently, we integrated the UBM scheme into the FreeBSD in a straightforward manner without any optimization, and hence we expect there is still much room for further optimizing the performance.

The UBM scheme also has the space overhead of maintaining ghost buffers in the kernel memory. The maximum size of the LRU stack to be maintained including ghost buffers is limited by the total number of buffers in the file system. Therefore, the space overhead due to ghost buffers is proportional to the difference between the total number of buffers in the system and the number of allocated buffers for other references. In the current

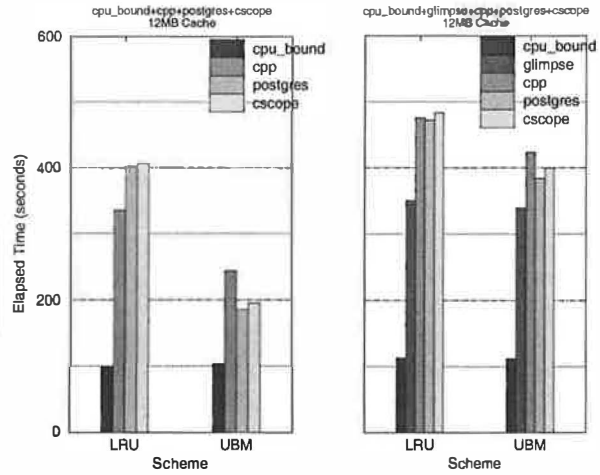


Figure 14: Run-time overhead of the UBM scheme.

implementation, each ghost buffer requires 13 bytes.

6 Conclusions and Future Work

This paper starts from the observation that the widely used LRU replacement scheme does not make use of regularities present in the reference patterns of applications, leading to degraded performance. The Unified Buffer Management (UBM) scheme is proposed to resolve this problem. The UBM scheme automatically detects sequential and looping references and stores the detected blocks in separate partitions of the buffer cache. These partitions are managed by appropriate replacement schemes based on the properties of their detected patterns. The allocation problem among the partitions is also tackled with the use of the notion of marginal gains.

To evaluate the performance of the UBM scheme, experiments were conducted using both trace-driven simulations with multiple application traces and an implementation of the scheme in the FreeBSD operating system. Both simulation and implementation results show that 1) the UBM scheme accurately detects almost all the sequential and looping references, 2) the UBM scheme manages sequential and looping-referenced blocks similarly to the OPT scheme, and 3) the UBM scheme shows substantial performance improvements increasing the buffer hit ratio by up to 57.7% (with an average increase of 29.2%) and reducing, in an actual implementation in the FreeBSD operating system, the elapsed time by up to 67.2% (with an average of 28.7%) compared to the LRU scheme, for the workloads we considered.

As future research, we are attempting to apply to other

references the Least Recently/Frequently Used (LRFU) scheme based on both recency and frequency factors rather than the LRU scheme, which is based on the recency factor only. Also, as automatic detection of sequential and looping references is possible, we are investigating the possibility of further enhancing performance through prefetching techniques that exploit these regularities as was attempted in [10] for informed prefetching and caching. Finally, we plan to extend the techniques presented in this paper to systems that integrate virtual memory and file cache management.

Acknowledgements

The authors are grateful to the anonymous reviewers for their constructive comments and to F. Douglass, our “shepherd”, for his guidance and help during the revision process. The authors also want to thank P. Cao for providing the ULTRIX trace used in the experiments.

References

- [1] J. T. Robinson and M. V. Devarakonda. Data Cache Management Using Frequency-Based Replacement. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 134–142, 1990.
- [2] E. J. O’Neil, P. E. O’Neil, and G. Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proceedings of the 1993 ACM SIGMOD Conference*, pages 297–306, 1993.
- [3] T. Johnson and D. Shasha. 2Q : A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on VLDB*, pages 439–450, 1994.
- [4] P. Cao, E. W. Felten, and K. Li. Application-Controlled File Caching Policies. In *Proceedings of the USENIX Summer 1994 Technical Conference*, pages 171–182, 1994.
- [5] V. Phalke and B. Gopinath. An Inter-Reference Gap Model for Temporal Locality in Program Behavior. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 291–300, 1995.
- [6] D. Lee, J. Choi, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies. In *Proceedings of the 1999 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 134–143, 1999.
- [7] E. G. Coffman, JR. and P. J. Denning. *Operating Systems Theory*. Prentice-Hall International Editions, 1973.
- [8] G. Glass and P. Cao. Adaptive Page Replacement Based on Memory Reference Behavior. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 115–126, 1997.
- [9] Y. Smaragdakis, S. Kaplan, and P. Wilson. EELRU: Simple and Effective Adaptive Page Replacement. In *Proceedings of the 1999 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 122–133, 1999.
- [10] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proceedings of the 15th Symposium on Operating System Principles*, pages 1–16, 1995.
- [11] D. Thiebaut, H. S. Stone, and J. L. Wolf. Improving Disk Cache Hit-Ratios Through Cache Partitioning. *IEEE Transactions on Computers*, 41(6):665–676, 1992.
- [12] C. Faloutsos, R. Ng, and T. Sellis. Flexible and Adaptable Buffer Management Techniques for Database Management Systems. *IEEE Transactions on Computers*, 44(4):546–560, 1995.
- [13] J. Choi, S. Cho, S. H. Noh, S. L. Min, and Y. Cho. Analytic Prediction of Buffer Hit Ratios. *IEEE Electronics Letters*, 36(1):10–11, 2000.
- [14] J. R. Spin. *Program Behavior: Models and Measurements*. New York: Elsevier-North Holland, 1977.

How to Build a Trusted Database System on Untrusted Storage

Umesh Maheshwari Radek Vingralek William Shapiro

STAR Lab, InterTrust Technologies Corporation, Santa Clara, CA 95054

{umesh, rvingral, shapiro}@intertrust.com

Abstract

Some emerging applications require programs to maintain sensitive state on untrusted hosts. This paper presents the architecture and implementation of a trusted database system, TDB, which leverages a small amount of trusted storage to protect a scalable amount of untrusted storage. The database is encrypted and validated against a collision-resistant hash kept in trusted storage, so untrusted programs cannot read the database or modify it undetectably. TDB integrates encryption and hashing with a low-level data model, which protects data and metadata uniformly, unlike systems built on top of a conventional database system. The implementation exploits synergies between hashing and log-structured storage. Preliminary performance results show that TDB outperforms an off-the-shelf embedded database system, thus supporting the suitability of the TDB architecture.

1 Introduction

Some emerging applications require trusted programs to run on untrusted hosts. For example, vendors of digital goods such as software and music need to control the use of their goods according to their contracts with the consumers. The contracts may be enforced by executing a trusted program on the consumer's computer or playing device [SBV95, IBM00, Xer00].

Often, trusted programs need to maintain some sensitive, persistent state. For example, under a pay-per-use contract, the program may verify and debit the consumer's account. Or, under a limited-use trial, the program may count and limit the number of times the good is used. The amount of such state may grow with the number of vendors, goods, and the types of contracts. Furthermore, the sensitive nature of the state makes it desirable to protect it from both tampering and accidental corruption. Therefore, the state should be stored in a scalable and trusted database system.

Although a trusted program runs on the client, it could maintain its database on a trusted server for best security. However, this may require frequent communication

between the trusted program and the server, which is constraining for devices with poor connectivity. Ideally, consumers should be able to use goods distributed on mass media or previously hoarded on their devices, even when they are disconnected from the network. Therefore, it is desirable to maintain the database on the client side.

The party hosting the database storage has the opportunity to alter its state for unauthorized benefits. For example, a consumer could save a copy of the local database, purchase some goods, then *replay* the saved copy, thus eliminating payments for the purchased goods.

It is difficult to secure a trusted program and its database because the hosting party ultimately controls the underlying hardware and the operating system. However, a number of emerging trusted platforms provide a processing environment that runs only trusted programs and resists reverse engineering and tampering. Such platforms employ a hardware package containing a processor, memory, and tamper-detecting circuitry [SPW98, KK99, Wav99, Dal00], or various techniques for software protection [Coh93, Auc96, CTL98]. However, these platforms do not provide trusted persistent storage in bulk because it is difficult to prevent read and write access to devices such as disk and flash memory from outside the trusted platform.

This paper presents the architecture and implementation of a trusted database system, TDB. By "trust" we mean *secrecy* (protection against reading from untrusted programs) and *tamper detection* (protection against writing from untrusted programs). An untrusted program cannot be prevented from tampering with the data, but such data fails validation when a trusted program reads it. This enables the trusted program to reject the data and perhaps refuse further operation.

TDB may also be used to protect a database stored at an untrusted server. Such a database may be used by client devices that do not have enough local storage. In this case, the user may have no incentive to tamper with the client device, so no explicit mechanisms may be required to provide a trusted platform on the client.

1.1 Basic Trust Management

TDB leverages a trusted processing environment and a small amount of trusted storage available on the platform. It provides secrecy by encrypting data with a key hidden in secret storage. It provides tamper detection by leveraging a small amount of tamper-resistant storage, as described below.

A common mechanism for validating data is to sign it with a secret key. However, signed data is vulnerable to replay attacks. The attack is easy because it does not require understanding the data; it works even when the data is encrypted. TDB resists replay attack by storing a collision-resistant hash of the database in tamper-resistant storage [MOV96]. When a trusted program writes and reads database objects, TDB updates and validates the database hash efficiently by maintaining a tree of hash values over the objects, as suggested by Merkle [Mer80].

TDB provides an option to use a tamper-resistant counter, which cannot be decremented, in place of generic tamper-resistant storage. After each database update, TDB increments the counter and generates a certificate containing the counter value and the database hash. The certificate is signed with the secret key and stored in untrusted storage.

1.2 Storage Management

To protect the state from accidental corruption, TDB provides standard database-system services such as crash atomicity, concurrent transactions, type checking, pickling, cache management, and index maintenance.

One might consider building a trusted database system by layering cryptography on top of a conventional database system. This layer could encrypt objects before storing them in the database and maintain a tree of hash values over them. This architecture is attractive because it does not require building a new database system. Unfortunately, the layer would not protect the metadata inside the database system. An attack could effectively delete an object by modifying the indexes. There could be some performance problems as well. For example, the database system could not maintain ordered indexes for range queries on encrypted data.

For these reasons, TDB applies hashing and encryption to a low-level data model, which protects data and metadata uniformly. It also enables TDB to maintain ordered indexes on data.

To protect the sensitive state from media failures such as disk crashes, TDB provides the ability to create backups and to restore valid backups. An attacker might fake a media failure and restore a backup to rollback the

state. To limit the extent of a rollback, it is desirable to make frequent backups and disallow restoring old backups. TDB facilitates this by providing incremental backups [HMF99].

We discovered and exploited the synergy between the functions mentioned above and log-structured storage systems [RO91]. Log-structured systems have a comprehensive and hierarchical location map, because all objects are relocatable. Embedding the hash tree in the location map allows an object to be validated as it is located. The checkpointing optimization defers and consolidates the propagation of hash values up the tree. Copy-on-write using the location map provides cheap snapshots, which enables incremental backups. Furthermore, the absence of fixed object locations makes it hard to link multiple updates to the same object, thus resisting some traffic-monitoring attacks.

Preliminary performance results show that TDB outperforms a system that layers cryptography on top of an off-the-shelf database system. The database overhead is dominated by I/O; encryption and hashing represent only 6% of the total overhead.

1.3 Outline

The rest of this paper is organized as follows. Section 2 specifies the infrastructure TDB requires and the service it provides. Section 3 describes the overall architecture of TDB. Sections 4 and 5 describe the integration of encryption and hashing in a low-level data model. Section 6 describes backup creation and restores. Sections 7 and 8 briefly describe the construction of database functions over the low-level data model. Section 9 gives preliminary performance results. Section 10 describes potential extensions to TDB. Section 11 compares TDB with related work. Section 12 draws some conclusions.

2 System Specification

This section specifies the infrastructure TDB requires and the service it provides to applications.

2.1 Required Infrastructure

TDB requires a trusted platform that provides the following, as shown in Figure 1:

- *Trusted processing environment*, which executes only trusted programs and protects the volatile state of an executing program from being read or modified by untrusted programs. The static image of a trusted program need not be secret.

- *Secret store*: a small amount (e.g., 16 bytes) of read-only persistent storage that can be read only by a trusted program.
- *Tamper-resistant store*: a small amount (e.g., 16 bytes) of writable persistent storage that can be written only by a trusted program. Alternatively, the tamper-resistant store may be a counter that cannot be decremented. In either case, we assume that the tamper-resistant store can be updated atomically with respect to crashes.

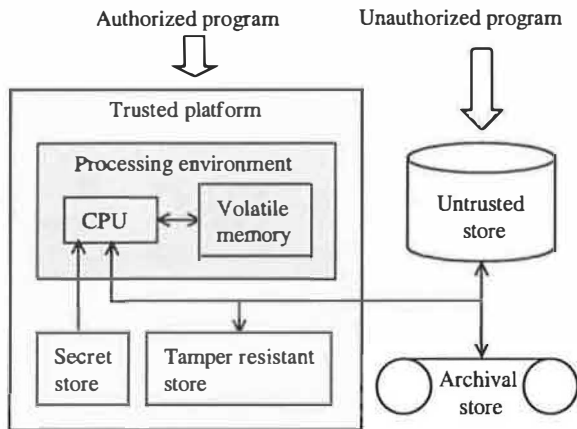


Figure 1: The trusted platform

The trusted platform may be a hardware package such as the IBM secure coprocessor [SPW98], which contains a processor, battery-backed SRAM, DRAM, and ROM. The ROM firmware loads only trusted programs using a hash supplied during the manufacturing process. The battery-backed SRAM is zeroed if tampering is detected, so it can serve as both secret and tamper-resistant store.

The infrastructure also provides an *untrusted store* to hold the database. It is persistent, allows efficient random access, and can be read and written by any program. This might be a disk, flash memory, or an untrusted storage server connected to the trusted platform.

An *archival store* is needed to recover from the failures of the untrusted store. It is also untrusted. It need not provide efficient random access to data, only input and output streams. It might be a tape or an ftp server. We assume its failures are independent of the untrusted store.

We assume that suitable steps are taken when tampering is detected. The exact nature of such steps is outside the scope of this paper.

2.2 Service Provided

We list the functions of TDB below.

Trusted storage: TDB provides tamper-detection and secrecy for bulk data. This includes resistance to replay attacks and attacks on metadata.

Partitions: An application may need to protect different types of data differently. For example, it may have no need to encrypt some data or to validate other data. TDB allows an application to create multiple logical partitions, each protecting data with its own cryptographic parameters:

- a secret key
- a cipher (an encryption algorithm), e.g., 3DES
- a collision-resistant hash function, e.g., SHA-1

Using appropriate parameters avoids unnecessary time and space overhead. Using different secret keys reduces the loss from the disclosure of a single key. This should not be confused with *access control* among trusted parties, which may be provided in a higher layer, if needed.

Atomic updates: TDB can update multiple pieces of data atomically with respect to fail-stop crashes such as power failures.

Backups: TDB can back up a consistent snapshot of a set of partitions and restore a backup after validation. Backups allow recovery from media corruption. TDB provides fast *incremental* backups, which contain only changes made since a previous backup.

Concurrent transactions: TDB provides serializable access to data from concurrent transactions. Unlike shared databases or file servers, TDB is not designed for simultaneous access by many users. Therefore, its concurrency control is geared to low concurrency. It employs techniques for reducing latency, but lacks sophisticated techniques for sustaining throughput.

Database size: TDB allows the database to scale with gradual performance degradation. It uses scalable data structures and fetches data piecemeal on demand. However, it relies on a cacheable working set for performance because its log-structured storage may destroy physical clustering. Another limitation is its no-steal buffering of dirty data, which does not scale to transactions with many modifications [GR93].

Objects: TDB stores abstract objects that the application can access without explicitly invoking encryption, validation, and pickling. TDB pickles objects using application-provided methods so the stored representation is compact and portable.

Collection and Indexes: TDB provides index maintenance over *collections* of objects. A collection is a set

of objects that share one or more indexes. An index provides scan, exact-match, and range iterators.

3 System Architecture

TDB is designed for use on personal computers as well as smaller devices. The architecture is layered, so applications can trade off functionality for smaller code size. In Figure 2, boxes represent modules and arrows represent dependencies between them. Dashed boxes represent infrastructural modules.

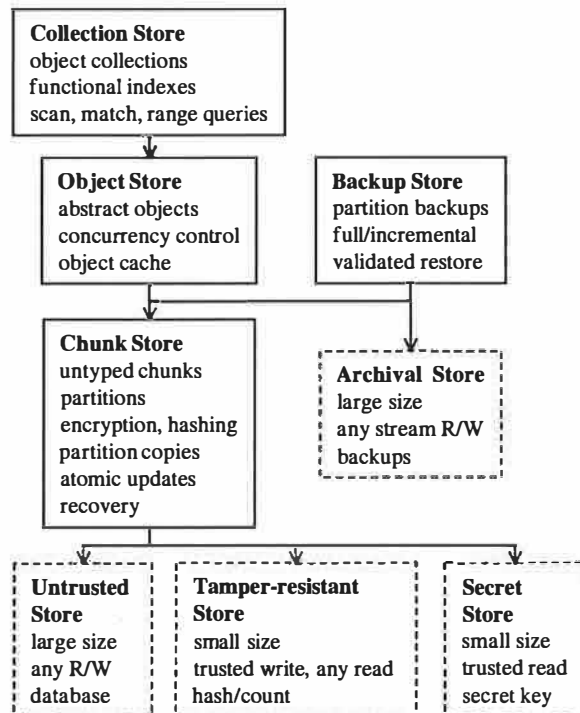


Figure 2: System architecture

The *chunk store* provides trusted storage for a set of named *chunks*. A chunk is a variable-sized sequence of bytes that is the unit of encryption and validation. (We expect chunk sizes between 100 bytes and 10 Kbytes.) All data and metadata from higher modules are stored as chunks. Chunks are logically grouped into partitions with separate cryptographic parameters. Partitions can be snapshot using the copy-on-write technique.

Chunks are stored in the untrusted store. The chunk store supports atomic updates of multiple chunks in the presence of crashes. It hides logging and recovery from higher modules. This architecture does not support logical logging, but the variable-sized chunks form a more compact log than fixed-sized pages.

The *backup store* creates and restores a set of partition backups. The chunk store and the backup store encapsulate secrecy and tamper-detection. This enables the higher modules to provide database management without worrying about trust.

The *object store* manages a set of named objects. It stores pickled objects in chunks—one or more objects per chunk. It keeps a cache of frequently-used or dirty objects. Caching data at this level is beneficial because the data is decrypted, validated, and unpickled. The object store also provides read transactional access to objects using read-write locking.

The *collection store* manages a set of named collections of objects. It updates the indexes on a collection as needed. Collections and indexes are themselves represented as objects.

This paper focuses on integrating trust with storage management in the chunk store and the backup store. It describes higher modules briefly to show that the chunk store is able to support them, and to explain a high-level performance benchmark we use.

4 Chunk Store: Single Partition

To simplify presentation, this section describes the chunk store as it would be in the absence of multiple partitions. Section 5 describes multiple partitions.

4.1 Specification

The chunk store manages a set of chunks named with unique ids. It provides the following operations:

- **Allocate()** returns *ChunkId*
Returns an unallocated chunk id.
- **Write(chunkId, bytes)**
Sets the state of chunkId to bytes, possibly of different size than the previous state. Signals if chunkId is not allocated.
- **Read(chunkId)** returns *Bytes*
Returns the last written state of chunkId. Signals if chunkId is not written.
- **Deallocate(chunkId)**
Deallocates chunkId. Signals if chunkId is not allocated.

Tamper Detection: In an idealized secret and tamper-proof chunk store, the operations listed above would be available only to trusted programs. Since tampering with the untrusted store cannot be prevented, the chunk store provides *tamper-detection* instead. It behaves like the tamper-proof store, except its operations may signal tamper detection if the untrusted store is tampered with.

Crash Atomicity and Durability: The write and deallocate operations are special cases of a *commit* operation. In general, a number of write and deallocate operations may be grouped into a single commit, which is atomic with respect to fail-stop crashes.

Allocated but unwritten chunks are deallocated automatically upon system restart. We have deliberately separated allocate and commit operations. An alternative is to allocate ids when new, unnamed chunks are committed. However, this alternative does not allow an application to store a newly-allocated chunk id in another chunk during the same commit operation, which may be needed for data integrity. Systems that swizzle application-provided references into persistent ids upon commit do not face this problem. However, the chunk store does not interpret application data chunks.

Concurrency Control: Operations are executed in a serializable manner. However, the chunk store is unaware of transactions. Allocate, read, and commit operations from different transactions may be interleaved.

4.2 Implementation Overview

This section gives an overview of the implementation; subsequent sections give further detail.

The chunk store writes chunks by appending them to a log in the untrusted store. As in other log-structured systems, chunks do not have static versions outside the log [RO91]. When a chunk is written or deallocated, its previous version in the log, if any, becomes obsolete.

The chunk store uses a *chunk map* to locate and validate the current versions of chunks. To scale to a large number of chunks, the chunk map is itself organized as a tree of chunks. Updates to the chunk map are buffered and written to the log occasionally. Updates lost upon a crash are recovered from the log.

Secrecy is provided by encrypting chunks with the key in the secret store. Tamper-detection is provided by creating a path of *hash links* from the tamper-resistant store to every current chunk version. We say there is a hash link from data x to y if x contains a hash of some data that includes y . If x is linked to y via one or more links using a collision-resistant hash function, it is computationally hard to change y without changing x or breaking a hash link [Mer80]. The hash links are embedded in the chunk map and the log.

Serializability of operations is provided through mutual exclusion, which does not overlap I/O and computation, but is simple and acceptable when concurrency is low.

4.3 Chunk Map

The chunk map maps a chunk id to a *chunk descriptor*, which contains the following information:

- status of chunk id: unallocated, unwritten, or written
- if written, current location in the untrusted store
- if written, expected hash value of chunk

Figure 3 shows the tree structure of the chunk map. The leaves are the chunks created by the applications of the chunk store; we call them *data chunks*. (These include chunks containing metadata of higher modules, for example, the indexing data of the collection store.) Each internal chunk, called a *map chunk*, stores a fixed-size vector of chunk descriptors. In the figure, each shaded slot is a chunk descriptor, and an arrow links the chunk containing the descriptor to the chunk described by the descriptor. The chunk at the top contains the descriptor of the root map chunk and some additional metadata needed to manage the tree; we call it the *leader chunk*. The descriptor of the leader chunk is retrieved at startup, as described later. The chunk store interprets map and leader chunks, but not data chunks.

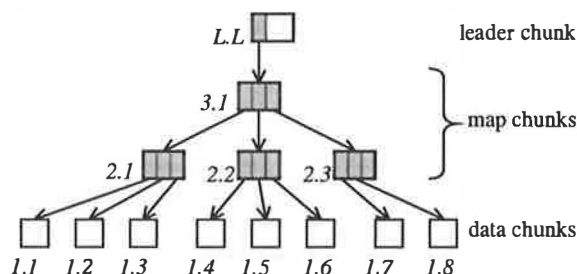


Figure 3: The chunk map

For uniformity of access and storage management, non-data chunks are also named using chunk ids. The id of a chunk encodes its *position* in the tree. The position comprises the height of the chunk in the tree and its rank from the left among the chunks at that height. In the figure, chunk ids are denoted as “*height.rank*”. As the tree grows, new chunks are added to the right and to the top, which preserves the positions of existing chunks. (The position of the leader does change, so it is given a reserved id instead.) Besides unifying access to chunks, this approach enables id-based navigation of the map without storing ids in the map explicitly.

4.4 Allocate Operation

Ids of deallocated data chunks are reused to keep the chunk map compact and conserve id space. Deallocated ids are linked through a free list embedded in the descriptors. The head of the list is stored in the leader.

As mentioned, id allocation is not persistent until the chunk is written (committed). Upon system restart, chunk ids that were previously allocated but not written are made available in the free list for re-allocation.

4.5 Read Operation

Given a chunk id c , its state may be located and validated by traversing the path of descriptors from the leader to c . For each descriptor in the path, the chunk state is found as follows. The encrypted state is read from the location stored in the descriptor. It is decrypted using the secret key. The decrypted state is hashed. If the computed hash does not match that stored in the descriptor, tamper detection is signaled.

For better performance, the chunk map keeps a cache of descriptors indexed by chunk ids. Also, the leader chunk is pinned in the cache. The cached data is decrypted, validated, and unpickled.

If the descriptor for c is not in cache, the read operation looks for the descriptor of c 's parent chunk. Thus, the read operation proceeds *bottom up* until it finds a descriptor in the cache. Then it traverses the path back down to c , reading and validating each chunk in the path. This approach exploits the validated cache to avoid validating the entire path from the leader to the specified chunk.

4.6 Commit Operation

The commit operation hashes and encrypts each chunk to be written, and writes the encrypted state to the log in the untrusted store. We refer to the set of chunks written as the *commit set*.

When a chunk c is written or deallocated, its descriptor is updated to reflect its new location, hash, or status. Conceptually, this changes c 's parent chunk d ; if d were also written out, its descriptor would be updated, and so on up to the leader, whose descriptor would be written to the tamper-resistant store. Instead, to save time and log space, the chunk store updates c 's descriptor in cache and marks it as dirty so it is not evicted. The bottom-up search during reads ensures that the stale descriptor stored in d is not used.

4.7 Checkpoint

When the cache becomes too large because of dirty descriptors, all map chunks containing dirty descriptors and their ancestors up to the leader are written to the log. This is done as a special commit operation called a *checkpoint*. In practice, checkpoints happen infrequently compared to regular commits. Other log-structured systems use similar checkpoints to defer and

consolidate updates to the location map [RO91]. The chunk store extends the optimization to propagating hash values up the chunk map.

The leader is written last during a checkpoint. We refer to the part of the log written before the leader as the *checkpointed log* and the part including and after the leader as the *residual log*. Figure 4 shows a simple example, where the log tail contains some data chunks, possibly written in multiple commits, a checkpoint containing the affected map chunks and the leader chunk, and some more data chunks. Arrows link chunks as in Figure 3.

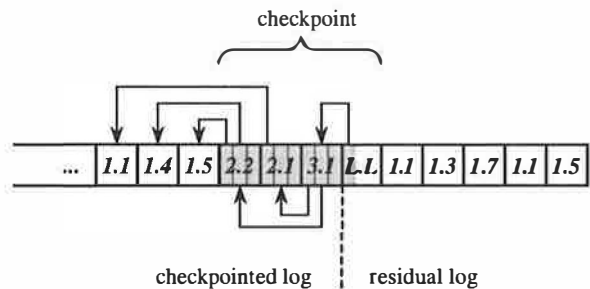


Figure 4: Checkpointing the chunk map

4.8 Recovery

A crash loses buffered updates to the chunk map, but they are recovered upon system restart by rolling forward through the residual log. Section 4.9 describes how the log is represented so the recovery procedure may find the sequence of chunks in the residual log.

For each chunk in the residual log, the recovery procedure computes the descriptor based on its location and hash, and puts the descriptor in the chunk-map cache. This procedure requires additional support from the commit operation to redo chunk deallocations and to validate the chunks in the residual log. This is described in the next two sections.

4.8.1 Chunk Deallocation

For each chunk to be deallocated, the commit operation writes a *deallocate chunk* to the log, which contains the id of the deallocated chunk.

Deallocate chunks are instances of *unnamed chunks*: they do not have chunk ids or positions in the chunk map. This is acceptable because they are used solely for recovery from the residual log and are always obsolete in the checkpointed log.

Like other chunks, unnamed chunks are encrypted with the secret key. They are also protected against tampering, as described in the next section. Otherwise, an at-

tack could cause a chunk to be un-deallocated. Or, an attack could replay the deallocation of a chunk id after it was re-allocated.

4.8.2 Validation of Residual Log

Although checkpointing defers the propagation of hash values up the chunk map, each commit operation must still update the tamper-resistant store to reflect the new state of the database. If the tamper-resistant store kept the hash of the leader and were updated only at checkpoints, the system would be unable to detect tampering with the residual log after a crash. We have implemented two approaches for maintaining up-to-date validation information in the tamper-resistant store.

4.8.2.1 Direct Hash Validation

The chunk store maintains a sequential hash of the residual log. The log hash is stored in the tamper-resistant store and updated after every commit. Upon recovery, the hash in the tamper-resistant store is matched against the hash computed over the residual log. This approach is illustrated in Figure 5.

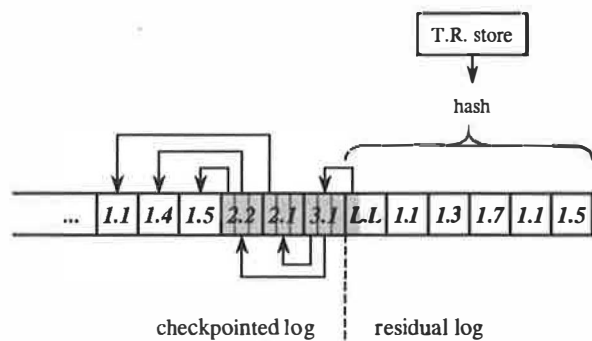


Figure 5: Tamper-resistant store contains database hash

A commit operation waits until the commit set is written to the untrusted store reliably before it updates the hash in the tamper-resistant store. Otherwise, a crash could leave the tamper-resistant store updated when the untrusted store is not, and cause validation to fail upon recovery. The update to the tamper-resistant store is the real commit point: If there is a crash during this update, the previous value stored in the tamper-resistant store is recovered, and the last commit set in the untrusted store is ignored. The commit operation returns after the tamper-resistant store is updated reliably.

Direct hash validation creates paths of hash links from the tamper-resistant store to all current chunk versions—in both the residual log and the checkpointed log. This is true because the tamper-resistant store is directly linked to all chunks in the residual log, which

includes the leader from the last checkpoint, and the leader is linked through the chunk map to all current chunk versions in the checkpointed log. Note that all unnamed chunks in the residual log are linked as well. Unnamed chunks in the checkpointed log are not linked, which is not a weakness because all such chunks are obsolete.

4.8.2.2 Counter-based validation

In this approach, upon each commit, a sequential hash of the commit set is stored in an unnamed chunk added to the log, called the *commit chunk*. The commit chunk is signed with the secret key. (The signature need not be publicly verifiable, so it may be based on symmetric-key encryption [MOV96].) An attack cannot insert an arbitrary commit set into the residual log because it will be unable to create an appropriately signed commit chunk. Replays of old commit sets are resisted by adding a count to the commit chunk that is incremented after every commit. Deletion of commit sets at the tail of the log is resisted by storing the current commit count in the tamper-resistant store. This approach is illustrated in Figure 6.

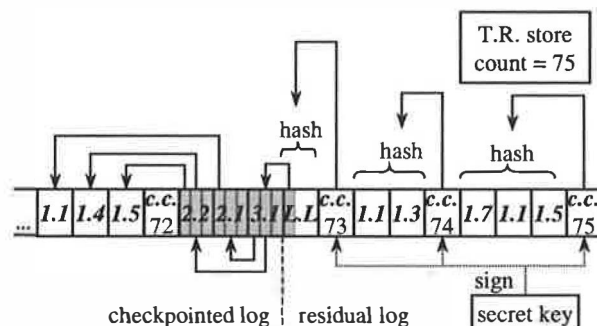


Figure 6: Tamper-resistant store contains commit count

A checkpoint is followed by a commit chunk containing the hash of the leader chunk, as if the leader were the only chunk in the commit set. The recovery procedure checks that the hash of each commit set in the residual log matches that stored in the commit chunk, and that the counts stored in the commit chunks form a sequence. Finally, the procedure compares the count in the last commit chunk with that in the tamper-resistant store. The hash-links created in this approach are similar to those in direct hash validation, except that the commit chunks are signed and linked from the tamper-resistant store through a sequence of numbers.

Counter-based validation has several advantages. First, the tamper-resistant counter is a weaker requirement than a generic tamper-resistant store. Provided the

counter cannot be decremented by *any* program, it does not need additional protection against untrusted programs. There is little incentive for untrusted programs to increment the counter because they would not be able to sign a commit chunk with the increased count.

Second, the commit count allows the system to tolerate bounded discrepancies between the tamper-resistant store and the untrusted store, if desired. For example, the system might allow the count in the tamper-resistant store, t , to be a little behind the last count in the untrusted store, u . This trades off security for performance. The security risk is that an attack might delete commit sets $t+1$ through u . The performance gain is that a commit operation need not wait for updating the count in the tamper-resistant store, provided $(u-t)$ is smaller than some threshold Δ_u . This is useful if the tamper-resistant store has high update latency. The system might also allow t to leap *ahead* of u by another threshold Δ_m . This admits situations where the untrusted store is written lazily (e.g., IDE disk controllers often flush their cache lazily) and the tamper-resistant store might be updated before the untrusted store. The only security risk is the deletion of at most Δ_m commit sets from the tail of the log.

A drawback of counter-based validation is that tamper detection relies on the *secrecy* of the key used to sign the commit chunk. Therefore, if a database system needed to provide tamper-detection but not secrecy, it would still need a secret store.

4.9 Log Representation

This section describes the structure of the data written to the log. The log consists of a sequence of chunks; we refer to the representation of a chunk in the log as a *version*.

4.9.1 Chunk Versions

Chunk versions are read for three different functions:

- Read operation, which uses the chunk id and the descriptor to read the current version.
- Log cleaning, which reads a *segment* of the checkpointed log sequentially.
- Recovery, which reads the residual log sequentially.

To enable sequential reading, the log contains information to identify and demarcate chunks. Each chunk version comprises a header followed by a body. The header contains the chunk id and the size of the chunk state. The header of an unnamed chunk contains a reserved id. Both the header and the body are encrypted with the secret key. Similarly, the hash of the residual log or a commit set covers both headers and bodies.

4.9.2 Head of Residual Log

The recovery procedure needs to locate the head and the tail of the residual log. The head of the residual log is the leader. Its location is stored in a fixed place, as in other log-structured storage systems. It need not be kept in tamper-resistant store: With direct hash validation, tampering with this state will change the computed hash of the residual log. With counter-based validation, it is possible for an attack to change the location to the beginning of another commit set. Therefore, the recovery procedure checks that the chunk at the stored location is the leader.

Because the location of the leader is updated infrequently—upon each checkpoint—storing it at a fixed location outside the log does not degrade performance. This location is written after the writes to the untrusted store and the tamper-resistant store have finished. Its update marks the completion of the checkpoint. If there is a crash before this update, the recovery procedure ignores the checkpoint at the tail of the log.

4.9.3 Tail of Residual Log

With direct hash validation, the location of the log tail may be stored in the tamper-resistant store along with the database hash. This works well because the write to the tamper-resistant store is the true commit point.

With counter-based validation, it is possible to infer the location of the tail from the log itself, as in conventional databases [GR93]. The last commit set in the log may have been corrupted in a crash. The hash stored in a commit chunk serves well as a checksum for the commit set. The recovery procedure stops when the hash of a commit set does not match the hash stored in the commit chunk.

4.9.4 Segments

The untrusted store is divided into fixed-size segments to aid cleaning, as in Sprite LFS [RO91]. The segment size is chosen for efficient reading and writing by the cleaner, e.g., on the order of 100 KB for disk-based storage. A segment is expected to contain many chunk versions. The size of a chunk version cannot exceed the segment size. A commit set may span multiple segments.

The log is represented as a sequence of potentially non-adjacent segments. Since the recovery procedure needs to read the residual log sequentially, segments in the residual log contain an unnamed *next-segment chunk* at the end, which contains the location of the next segment.

4.9.5 Log Cleaning

The log cleaner reclaims the storage of obsolete chunk versions and compacts the storage to create empty segments. It selects a segment to clean and determines whether each chunk version is current by using the chunk id in the header to find the current location in the chunk map. It then commits the set of current chunks, which rewrites them to the end of the log [BHS95].

The set of steps from selecting a segment to committing the current chunks happens atomically with respect to externally invoked operations. The cleaner may be invoked synchronously when space is low, but it is mostly invoked asynchronously during idle periods.

The cleaner does not clean segments in the residual log, because that would destroy the sequencing of the residual log. This also resolves what the cleaner should do with unnamed chunks, because they are always obsolete in the checkpointed log. For performance reasons, the cleaner selects segments with low utilization. Details on the utilization metric and the maintenance of this information are beyond the scope of this paper.

The cleaner need not validate the chunks read from the segment provided the commit operation for rewriting current chunks does *not* update the hash values stored in chunk descriptors. If the hashes are recomputed and updated, as they would be in a regular commit, the cleaner must validate the current chunks; otherwise, the cleaner might launder chunks modified by an attack. Because of its simplicity, we have implemented the second, less efficient, approach.

5 Chunk Store: Multiple Partitions

This section describes extensions to the chunk store that provide multiple partitions and partition copies. Multiple partitions enable the use of different cryptographic parameters for different types of data. Partition copies enable fast backups.

5.1 Specification

The chunk store manages a set of named partitions, each containing a set of named chunks. A chunk id comprises the chunk position, as before, and the id of the containing partition. (A chunk in one partition may have the same position as another chunk in another partition.) The chunks in a partition are protected with the parameters associated with it.

The following partition operations are provided:

- **Allocate()** returns `PartitionId`
Returns an unallocated partition id.
- **Write(partitionId, secretKey, cipher, hashFunction)**
Sets the state of partitionId to an empty partition with the specified cryptographic parameters.
- **Write(partitionId, sourcePid)**
Copies the current state of sourcePid to partitionId. Each chunk in sourcePid is logically duplicated in partitionId at the same position.
- **Diff(oldPid, newPid)** returns `set<ChunkPosition>`
Returns a set containing chunk positions whose state is different in newPid and oldPid.
- **Deallocate(partitionId)**
Deallocates partitionId and all of its copies, and all chunks in these partitions.

Furthermore, the chunk allocate operation requires the id of the partition in which the chunk is to be created. A commit operation may include a number of write and deallocate operations on both partitions and chunks. This makes it possible, for example, to store the id of a newly-written partition into a chunk in an existing partition in one atomic step.

The next few sections describe how the extended specification is implemented.

5.2 Multi-partition Chunk Map

Figure 7 shows the structure of the multi-partition chunk map. Each written partition has a *position map*, which maps a chunk position in the partition to a descriptor. This map is like the single-partition map described in Section 4.3. The map chunks in the position map of partition P belong to P : their partition id is P and they are protected using P 's cryptographic parameters. In the figure, chunk ids are denoted as *partition:position*.

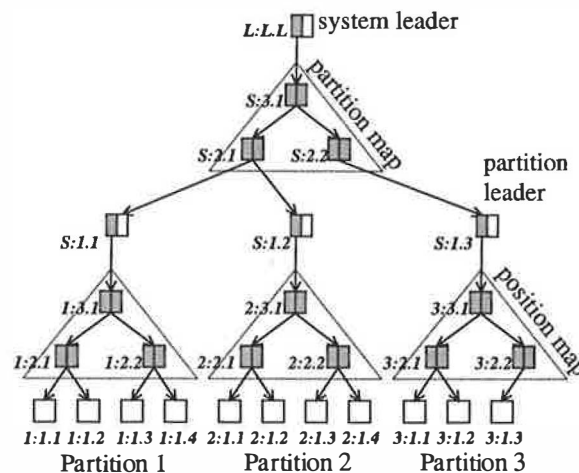


Figure 7: Multi-partition chunk map

The leader chunk for a partition contains information needed to manage the position map, as before, and the cryptographic parameters of the partition, including the secret key. The *partition map* at the top maps a partition id to the partition leader. This map is managed like the position map of a special partition, called the *system partition*, which has a reserved id denoted S in the figure. The partition leaders are the data chunks of the system partition and are protected using the cryptographic parameters of the system partition. Many partition operations such as allocating a partition id or reading a partition leader translate into chunk-level operations on the system partition.

Chunks in the system partition and the system leader are protected using a fixed cipher and hash function that are considered secure, such as 3DES and SHA-1 [MOV96]. They are encrypted with the key in the secret store. Thus, secrecy is provided by creating a path of *cipher links* from the secret store to every current chunk version. We say that there is a cipher link from one piece of data to another if the second is encrypted using a key stored in the first.

5.3 Partition Copies and Diffs

To copy a partition P to Q , the chunk store copies the contents of P 's leader to Q 's leader. Thus, Q and P share both map and data chunks, and Q inherits the cryptographic parameters of P . Thus, partition copies are cheap in space and time.

When chunks in P are updated, the position map for P is updated, but that for Q continues to point to the chunk versions at the time of copying. The chunks of Q can also be modified independently of P , but the common use is to create a read-only copy, called a *snapshot*.

The chunk store diffs two partitions by traversing their position maps and comparing the descriptors of the corresponding chunks. Commonly, diffs are performed between two snapshots of the same partition.

5.4 Log Representation

A commit set may contain chunks from different partitions. A chunk body is encrypted with the secret key and cipher of its partition. However, chunk headers are encrypted with the system key and cipher, so that cleaning and recovery may decrypt the header without knowing the partition id of the chunk.

The system leader is the head of the residual log, so it is linked from the tamper-resistant store. The residual log is hashed using the system hash function. Thus, each chunk in a commit set is hashed twice: once with its partition-specific hash function to update the chunk

descriptor, and once with the system hash function to update the log hash. In principle, the log hash could be computed over the partition-specific hashes of chunk bodies. However, a weak partition hash function could then invalidate the use of the log hash as a checksum for recovery (see Section 5.4). For simplicity, and because hashing is relatively fast, we chose to keep the hashes separate.

5.5 Cleaning and Recovery

Checking whether a chunk version is current is complicated by partition copies. A chunk header contains the id of the partition P to which it belonged when the chunk was written. Even if the version is obsolete in P , it may be current in some direct or indirect copy of P . Therefore, each partition leader stores the ids of its direct copies and the cleaner checks for current-ness in the copies, recursively. The process would be more complex had it not been that the deallocation of a partition deallocates the partition's copies as well.

Suppose the cleaner rewrites a chunk version identified as $P:x$ that is current only in partitions Q and R . The commit procedure updates the descriptors for $Q:x$ and $R:x$ in the cache. Further, in order that the recovery procedure is able to identify the chunk correctly, the cleaner appends an unnamed *cleaner chunk*, which specifies that the chunk is current in both Q and R .

6 Backup Store

The backup store creates and restores *backup sets*. A backup set consists of one or more *partition backups*. The backup store creates backup sets by streaming backups of individual partitions to the archival store and restores them by replacing partitions with the backups read from the archival store.

6.1 Backup Consistency

The backup store guarantees consistency of backup creation and restore with respect to other chunk store operations. Instead of locking each partition for the entire duration of backup creation, the backup store creates a consistent snapshot of the source partitions using a single commit operation. It then copies the snapshots to archival storage in the background. We assume that restores are infrequent, so it is acceptable to stop all other activity while a restore is in progress.

6.2 Backup Representation

Partition backups may be *full* or *incremental*. A full partition backup contains all data chunks of the partition. An incremental backup of a partition is created

with respect to a previous snapshot, the *base*, and contains the data chunks that were created, updated, or de-allocated since the base snapshot. Backups do not contain map chunks since chunk locations in the untrusted store are not needed. Chunks in a backup are represented like chunk versions in the log.

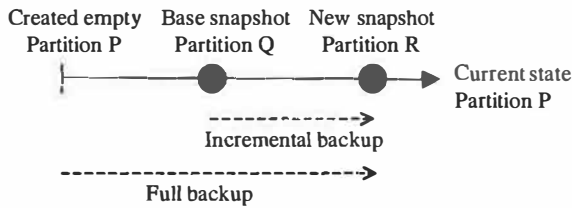


Figure 8: Full and incremental backups

A partition backup contains a *backup descriptor*, a sequence of chunk versions, and a backup signature. The backup descriptor contains the following (illustrated using partition ids from Figure 8):

- id of source partition (*P*)
- id of partition snapshot used for this backup (*R*)
- id of base partition snapshot (*Q*, if incremental)
- backup set id (a random number assigned to the set)
- number of partition backups in the backup set
- partition cipher and hasher
- time of backup creation

The representation of partition backups is illustrated below. Here, H_s denotes the system hash function, H_p denotes the partition hash function, E_s denotes system cipher using the system key, and E_p denotes the partition cipher using the partition key.

```
PartitionBackup ::=
    Es(BackupDescriptor)
    ( Es( ChunkHeader ) Ep(ChunkBody) ) *
    BackupSignature
    Checksum
```

```
BackupSignature ::=
    Es(Hs(BackupDescriptor Hp((ChunkId ChunkBody)*)))
```

The backup signature binds the backup descriptor with the chunks in the backup and guarantees integrity of the partition backup. The unencrypted checksum allows an external application to verify that the backup was written completely and successfully.

6.3 Backup Restore

The backup store restores a backup by reading a stream of one or more backup sets from the archival store. The backup store restores one partition at a time, enforcing the following constraints:

- Incremental backups are restored in the same order as they were created, with no missing links in between.

This is enforced by matching the base partition id in the backup descriptor against the id of the previous restored snapshot for the same partition.

- If a partition backup is restored, the remaining partition backups in the same backup set must also be restored. This is enforced by matching the number of backups with a given set id against the set size recorded in backup descriptors.

After reading the entire backup stream, the restored partitions are atomically committed to the chunk store. Backup restores require approval from a trusted program, which may deny frequent restoring or restoring of old backups.

7 Object Store

The *object store* adds safety against errors in application programs. It provides type-safe and transactional access to a set of objects. An object is the unit of typed data accessed by the application. The object store implements two-phase locking on objects and breaks deadlocks using timeouts. Transactions acquire locks in either shared or exclusive mode. We chose not to implement granular or operation-level locks because we expect only a few concurrent transactions. The object store keeps a cache of frequently-used or dirty objects. Caching data at this level is beneficial because the data is decrypted, validated, and unpickled.

The object store could store one or more pickled objects in each chunk. We chose to store each object in a different chunk because it results in a smaller volume of data that must be encrypted, hashed, and written to the log upon a commit. In addition, the implementation of the cache is simplified since no chunk can contain both committed and uncommitted objects. On the other hand, storing each object in a different chunk destroys inter-object clustering and increases the database size due to per-chunk overhead (see Section 9.3). Because we expect much of the working set to be cached, the lack of inter-object clustering is not important.

8 Collection Store

The *collection store* provides applications with indexes on *collections* of objects. A collection is a set of objects sharing one or more indexes. Indexes can be dynamically added and removed from each collection. Collections and indexes are themselves represented as objects.

The collection store supports *functional indexes* that use keys extracted from objects by deterministic functions [Hwa94]. The use of functional indexes allows us to avoid a separate data definition language for the database schema. Indexes are maintained automatically as

objects are updated. Indexes may be unsorted or sorted, which is possible because the objects are decrypted.

9 Performance

In this section we describe preliminary performance measurements. First, we present the performance on chunk and backup store operations based on several micro-benchmarks. Then we compare the performance an off-the-shelf database system and TDB using a higher-level benchmark.

9.1 Platform

Performance was evaluated on a 450 MHz Pentium PC with 128 MB of RAM, running the Windows NT 4.0 operating system. TDB is written in C++.

The untrusted store was implemented as an NTFS file on a hard disk with 9 ms average seek and 7200 rpm (4 ms average rotational latency). Using a raw disk partition would be more efficient, but we do not expect the users of TDB to provide one. The total size of TDB caches (including the object cache and the chunk-map cache) was set to 4 Mbytes.

The tamper-resistant store was emulated with an NTFS file on another hard disk to avoid interference with accesses to the untrusted store. This disk has 12 ms average seek and 5200 rpm (6 ms average rotational latency). The access time is similar to that for writing EEPROM, 5 ms [Inf00].

We used counter-based validation and allowed the count in the tamper-resistant store to lag behind that in untrusted store by $\Delta_{ur} = 5$. The tamper-resistant store is flushed only once is Δ_{ur} commits. The untrusted store is flushed upon every commit and we set Δ_{tu} to 0.

9.2 Micro-benchmarks

This section presents the performance of basic cryptographic, disk, chunk store and backup store operations.

9.2.1 Cryptographic and Disk Operations

Encryption: We used 3DES in CBC mode for the system partition, which has a measured bandwidth of 2.5 MB/s (0.4 μ s per byte). We used DES in CBC mode for other partitions; the measured bandwidth is 7.2 MB/s (0.14 μ s per byte). There are other, more secure, algorithms that run faster than DES [MOV96].

Hashing: We used SHA-1. The measured bandwidth is 21.1 MB/s (0.05 μ s per byte). Additionally, the “finalization” of a hash value has a fixed overhead of 5 μ s.

Store latency: While the disk specs provide average latency, the measured latency varies widely based on the position of disk head. Furthermore, the latency of the NTFS flush operation for files larger than 512 bytes is doubled because it writes file metadata separately. We measured write latencies of 10 ms to 20 ms for small files and 25 ms to 40 ms otherwise. Therefore, we shall focus on the computational overhead and denote the latencies of the untrusted and tamper-resistant store symbolically as l_u and l_r .

Store bandwidth: The measured bandwidth, b_u , of reading or writing the NTFS file implementing the untrusted store varies between 3.5 and 4.7 MB/s.

9.2.2 Chunk Store Operations

We repeated each operation 10 times and found that the computational overhead does not vary much, typically deviating less than 2%.

Allocate chunk id: This operation does not change the persistent state. The average latency is 6 μ s.

Write chunks + commit: We committed sets of 1 to 128 chunks of sizes 128 bytes to 16 KB per chunk, which covers the range we expect. The computational latency, measured using linear regression, is 132 μ s + 36 μ s per chunk + 0.24 μ s per byte of cumulative chunk size. The fixed overhead comes largely from processing the commit chunk (pickling, encrypting, hashing, etc.), the per-chunk overhead from processing the chunk header and finalizing the chunk’s hash value, and the per-byte overhead from encryption and hashing the chunk bodies. The I/O overhead is $l_u + l_r/\Delta_{ur} + 1/b_u$ per byte, which usually dominates the computational overhead.

Read chunk: If the chunk descriptor is cached, the computational latency of reading a chunk is 47 μ s + 0.18 μ s per byte of chunk size. The fixed overhead comes largely from processing the chunk header and finalizing the hash, and the per-byte overhead from decryption and hashing. The I/O overhead is $l_u + 1/b_u$ per byte. If the descriptor is not cached, the read operation reads in parental map chunks up to one whose descriptor is cached. In our experiments, each map chunk has 64 descriptors and has a size of 1.5 KB.

Write partition + commit: The computational latency of committing a new partition is 223 μ s. The computational latency of copying a partition is 386 μ s, regardless of the number of chunks in the source partition, owing to our use of the copy-on-write technique.

9.2.3 Backup Store Operations

We benchmarked only backup creation, we assume that backup restore performance is not critical.

Partition backup: We used 512 byte chunks. The computational latency to create an incremental backup of a partition is $675 \mu\text{s} + 9 \mu\text{s}$ per chunk in the backed up partition + $278 \mu\text{s}$ per updated chunk. The fixed overhead comes mostly from creating the partition snapshot and processing the backup descriptor and signature. The overhead per chunk in the backed up partition comes from diff-ing the snapshot of the backed up partition against the base snapshot. The overhead per updated chunk comes from copying the chunk.

The size of a backup determines the I/O overhead for writing it. The size of an incremental backup is 456 B + 528 B per updated chunk, which may be significantly less than the size of a full backup.

9.3 Space Overhead

The chunk descriptor, header, and padding add an overhead of about 52 bytes for chunks encrypted using an 8-byte block cipher. The additional overhead per chunk due to the chunk map is small because the fanout degree of the tree is large (64). Obsolete chunk versions in the log add additional overhead. When cleaning in idle periods, the space utilization may be kept as high as 90% with reasonable performance [BHS95].

9.4 Code Complexity

Figure 9 gives the complexity of TDB in terms of number of semicolons in C++ code.

Module	semicolons
Collection store	1,388
Object store	512
Backup store	516
Chunk store	2,570
Common utilities	1,070
TOTAL	6,056

Figure 9: TDB code complexity

9.5 Performance Comparison

In this section, we compare the performance of a system using either TDB or an off-the-shelf embedded database system, which we shall call XDB. The XDB-based system layers cryptography on top of XDB. We configured both systems to use the same cryptographic parameters, cache size, and frequency of flushing the tamper-resistant store.

9.5.1 Workload

We measured the performance on a benchmark that models two operations related to vending digital goods:

- **Bind:** A vendor binds three alternative contracts to a digital good.
- **Release:** A consumer releases the digital good selecting one of the three contracts randomly.

The benchmark first creates 30 collections for different object types. Each collection has one to four indexes. The benchmark loads the cache before executing an experiment. The experiment consists of 10 consecutive bind or release operations. Figure 10 gives the number of database operations executed in each experiment.

	read	update	delete	add	commit
release	781	181	10	41	96
bind	1732	733	10	220	292

Figure 10: Number of database operations.

9.5.2 Comparison Results

We repeated each experiment 10 times. Figure 11 shows the average times for the release and bind experiments, the part spent in the database system, and the part thereof spent in commit, which is the major overhead.

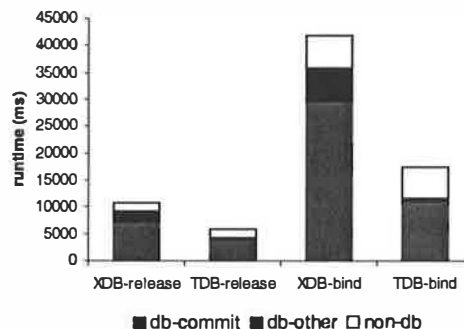


Figure 11: Runtime comparison

TDB outperformed XDB, primarily because of faster commits, but also in the remaining database overhead. We believe that XDB performs multiple disk writes at commit.

The stored size of XDB after running the release experiment was 3.8 MB. The stored size of TDB was 4.0 MB, based on 60% maximum log utilization.

9.5.3 TDB Performance Analysis

Here, we analyze the performance of the release experiment. Figure 12 breaks down the TDB overhead by

module. The time reported for each module excludes nested calls to other reported modules. The figure gives the average time (μ), the standard deviation (σ), and percentage of total (%).

module	μ (ms)	σ (ms)	%
DB TOTAL	4209	484	100
collection store	162	0	4
object store	85	0	2
chunk store	61	1	1
encryption	157	1	4
hashing	93	5	2
untrusted store read	8	0	0
untrusted store write	3353	164	81
tamper-resistant store	229	46	6

Figure 12: TDB runtime analysis

The overhead is dominated by writes to the untrusted store. The experiment flushed the untrusted store 96 times and the tamper-resistant store 19 times. The overhead of writing to the tamper-resistant store may vary significantly depending on the device and the frequency of flushes. There was no checkpoint or log cleaning during the experiment. (In the bind experiment, log cleaning took a total of 1030 ms.)

The overhead of encryption and hashing is only 6% of the database overhead. The effective bandwidths of encryption and hashing are 6.5 MB/s and 20.6 MB/s, which are close to the peak bandwidths reported in Section 9.2.1.

10 Potential Extensions

The current design of TDB has a number of limitations. Below we describe extensions to address them.

Untrusted storage on servers: TDB may be used to protect a database stored at an untrusted server. This application of TDB may benefit from additional optimizations for reducing network round-trips to the untrusted server, such as batching reads and writes.

Trusted paging. The current design assumes that the entire runtime, volatile state of a trusted program is protected by the trusted processing environment. TDB limits its volatile state by controlling its cache size, but this limit is not hard. Therefore, some volatile state may have to be paged out to untrusted storage. This problem may be solved by using a page fault handler to store encrypted and validated pages in the chunk store.

Steal buffer management. Currently, modified objects must remain in the cache until their transaction commits, which may degrade the security and performance of large transactions. Evicting dirty objects would require writing them to the log. This requires additional support in the chunk store.

Logical logging. Logical logging may reduce the volume of data that must be encrypted, hashed, and written to the untrusted store. The chunk store uses logical logging for some operations (for example, deallocation of chunks), but it does not allow higher modules to specify operations that should be logged logically.

11 Related Work

There are many systems aimed at providing secure storage. TDB differs from most of them because of its unique trust model.

In another paper at this conference, Fu et al. describe a read-only file system that may be stored in untrusted servers [FKM00]. A hash tree is embedded in the inode hierarchy. The trusted creator signs the root hash with the time of update and expiration. This system is not designed to handle frequent updates or updates to individual file blocks in the untrusted server.

Techniques for securing audit logs stored on weakly-protected hosts are suitable for securing append-only data that is read infrequently and sequentially by a trusted computer [BY97, SK98]. They employ a linear chain of hash values instead of a tree. When the data needs to be read, it is validated by recomputing the hash over the entire log. These techniques are not suitable for a database system such as ours, which requires frequent and random read-write access to data.

Blum et al. considered the problem of securing various data structures in untrusted memory using a hash tree rooted in a small amount of trusted memory [BEG+91]. This work does not address storage management for persistent data.

Some systems provide secure storage by dispersing data onto multiple hosts, with the expectation that at least a certain fraction of them (for example, two-thirds) will be honest. The data may be replicated as-is for time efficiency [CL99], or it might be encoded to reduce the cumulative space overhead [Rab89, Kra93, GGJ+97]. Read requests are broadcast to all machines and the data returned is error corrected. This approach provides *recovery* from tampering, not merely tamper detection. However, it relies on more trusted resources than are available to TDB. The expectation of an honest quorum is based on the assumption that, under normal operation, the hosts are weakly protected but not hostile, so

the difficulty for a hostile party to take over k hosts increases significantly with k .

Our use of log-structured storage builds on a previous work on log-structured storage systems [RO91, JKH93]. The Shadows database system is log structured and provides snapshots [Ylo94]. Otherwise, there has been little interest in log-structured database systems, perhaps because of the need to keep large sets of data physically clustered or to keep the log compact using logical logging.

12 Conclusions

We have presented a trusted database system that leverages a trusted processing environment and a small amount of trusted storage to extend tamper-detection and secrecy to a scalable amount of untrusted storage. The architecture integrates encryption and hashing with a low-level data model, which protects data and meta-data uniformly. The model is powerful enough to support higher-level database functions such as transactions, backups, and indexing.

We found that log-structured storage is well suited for building such a system. The implementation is simplified by embedding a hash tree in the comprehensive location map that is central to log-structured systems: objects can be validated as they are located. The check-pointing optimization defers and consolidates the propagation of hash values up the tree. Because updates are not made in place, a snapshot of the database state can be created using copy-on-write, which facilitates incremental backups.

We measured the performance of TDB using micro-benchmarks as well as a high-level workload. The database overhead was dominated by writes to the untrusted store and the tamper-resistant store, which may vary significantly based on the types of devices used. The overhead of encryption and hashing was only 6% of the total. On this workload, TDB outperformed a system that layers cryptography on an off-the-shelf embedded database system, while also providing more protection. This supports the suitability of the TDB architecture.

Acknowledgements

Olin Sibert and Susan Owicki motivated us to work on this problem. Various members of STAR Lab and InterTrust provided useful comments and help with performance measurement. Our shepherd, Frans Kaashoek, guided us in improving the presentation.

References

- [Auc96] D. Aucsmith. Tamper resistant software: an implementation. In *Proc. International Workshop on Information Hiding*, Lecture Notes in Computer Science, Vol. 1174, Cambridge, UK, 1996, pp. 317-333.
- [BEG+91] M. Blum, W. Evans, P. Gemmel, S. Kannan, and M. Naor. Checking the correctness of memories. In *Proc. IEEE Conf. on Foundations of Computer Science*, San Juan, Puerto Rico, 1991, pp. 90-99.
- [BHS95] T. Blackwell, J. Harris, and M. Seltzer. Heuristic cleaning algorithms in log-structured file systems. In *Proc. USENIX Technical Conference*, New Orleans, LA, 1995, pp. 249-264.
- [BY97] M. Bellare and B. Yee. Forward integrity for secure audit logs. *Technical Report*, Computer Science and Engineering Department, University of California at San Diego, 1997.
- [Coh93] F. Cohen. Operating system protection through program evolution. In *Computers & Security*, 12(6), Oxford, 1993.
- [CL99] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proc. Symposium on Operating Systems Design and Implementation*, New Orleans, LA, 1999, pp. 173-186.
- [CTL98] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proc. ACM Principles of Programming Languages*, San Diego, CA, 1998, pp. 184-196.
- [Dal00] Dallas Semiconductor secure microcontroller family, <http://www.dalsemi.com/products/micros/secure.html>, August 2000.
- [FKM00] K. Fu, F. Kaashoek, and D. Mazieres. Fast and secure distributed read-only file system. To appear in *Proc. Symposium on Operating Systems Design and Implementation*, San Diego, CA, 2000.
- [GGJ+97] J. Garay, R. Gennaro, C. Jutla, and T. Rabin. Secure distributed storage and retrieval. In *Proc. Intl. Workshop on Distributed Algorithms*, Berlin, Germany, 1997, pp. 275-289.
- [GR93] J. Gray and A. Reuter. *Transaction processing: concepts and techniques*. Morgan Kaufmann Publishers, 1993.
- [HMF+99] N. Hutchinson, S. Manley, M. Federwisch, G. Harris, D. Hitz, S. Kleiman, S. O'Malley. Logical vs.

- physical file backup. In *Proc. Symp. on Operating System Design and Implementation*, New Orleans, LA, 1999, pp. 239-249.
- [Hwa94] D. Hwang. Function-based indexing for object-oriented databases. *PhD thesis*, Massachusetts Institute of Technology, 1994.
- [Inf00] Infineon Technologies. Eurochip II—SLE 5536, http://www.infineon.com/cgi/ecrm.dll/ecrm/scripts/prod_ov.jsp?oid=14702&cat_oid=-8233, August 2000.
- [Int00] InterTrust Technologies Corp. Digital rights management. <http://www.intertrust.com/de/index.html>, August 2000.
- [IBM00] IBM. Cryptolope technology, <http://www.software.ibm.com/security/cryptolope>, August 2000.
- [JKH93] W. Jonge, M. F. Kaashoek, W. Hsieh. The logical disk: a new approach to improving file systems. In *Proc. ACM Symposium on Operating Systems Principles*, Asheville, NC, 1993, pp. 15-28.
- [KK99] O. Kommerling and M. Kuhn. Design principles for tamper-resistant smartcard processors. In *Proc. USENIX Workshop on Smartcard Technology*, Chicago, IL, 1999.
- [Kra93] H. Krawczyk. Distributed fingerprints and secure information dispersal. In *Proc. ACM Symp. on Principles of Distributed Computing*, Ithaca, NY, 1993, pp. 207-218.
- [Mer80] R. Merkle. Protocols for public key cryptosystems. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, 1980, pp. 122-134.
- [MOV96] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of applied cryptography*. CRC Press, 1996.
- [Rab89] T. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2), 1989, pp. 335-348.
- [RO91] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *Proc. ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, 1991, pp. 1-15.
- [SBV95] O. Sibert, D. Bernstein, and D. Van Wie. DigiBox: a self protecting container for information commerce. In *Proc. USENIX Conference on Electronic Commerce*, New York, NY, 1995, pp. 171-186.
- [SK98] B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. In *Proc. USENIX Security Symposium*, San Antonio, TX, 1998, pp. 52-62.
- [SPW98] S. Smith, E. Palmer, and S. Weingart. Using a high-performance, programmable secure coprocessor. In *Proc. Intl. Conf. on Financial Cryptography*, Anguilla, British West Indies, 1998.
- [Wav99] Wave Systems Corp. The Embassy e-commerce system. <http://www.wave.com/technology/Embassywhitepaper.pdf>, August 2000.
- [Xer00] ContentGuard, Rights management from Xerox, <http://www.contentguard.com>, August 2000.
- [Ylo94] T. Ylonen. Shadow paging is feasible. *Licentiate's thesis*, Helsinki University of Technology, 1994.

End-to-end authorization

Jon Howell*
*Consystant Design
Technologies*

David Kotz
*Department of Computer Science
Dartmouth College*

Abstract

Many boundaries impede the flow of authorization information, forcing applications that span those boundaries into hop-by-hop approaches to authorization. We present a unified approach to authorization. Our approach allows applications that span administrative, network, abstraction, and protocol boundaries to understand the end-to-end authority that justifies any given request. The resulting distributed systems are more secure and easier to audit.

We describe boundaries that can interfere with end-to-end authorization, and outline our unified approach. We describe the system we built and the applications we adapted to use our unified authorization system, and measure its costs. We conclude that our system is a practical approach to the desirable goal of end-to-end authorization.

1 Introduction

As systems grow more complex, they are often grown by affixing one system to another using some form of gateway to bridge boundaries between the systems. The boundaries can take several forms; we discuss four in this paper.

When we assemble systems in this way, frequently the authorization information available at the client system cannot be translated to the terms of authorization at the server system. As a result, the gateway often ends up making access-control decisions on behalf of the server system, and the server system is ignorant of any authorization information beyond a blind trust in the gateway. Our end-to-end authorization system remedies this situation.

2 Goals

Saltzer et al. describe a general principle for computer engineering: implement end-to-end semantics

*Supported by a research grant from the USENIX Association. This work represents part of Jon's Ph.D work at Dartmouth College. Jon can be reached at jonh@alum.dartmouth.org.

to achieve correctness, and only implement hop-by-hop semantics to boost the performance of the end-to-end implementation [19]. Voydock and Kent argue for end-to-end security measures when the hops are between network routers [24]. The same principle applies to authorization semantics when the hops are between gateways that span administrative boundaries, network scales, levels of abstraction, or protocol boundaries. End-to-end authorization makes systems more secure by reducing the number of programs that make access-control decisions, by giving those programs that do control access more thorough information, and by providing more useful audit trails. In this section, we illustrate four kinds of boundaries in distributed systems that impede the flow of authorization information from one end of a system to another. We discuss how, by giving clients and servers the ability to form and verify proofs, our unified system can support end-to-end authorization through the gateways that span these boundaries.

2.1 Spanning administrative domains

Administrative boundaries frequently interfere with end-to-end authorization. The conventional approach to authorization involves authenticating the client to a local, administratively-defined user identity, then authorizing that user according to an access-control list (ACL) for the resource. When resources are to be shared across administrative boundaries, this scheme fails because the server has no local knowledge of the recipient's identity.

Typical solutions to this problem involve authenticating the remote user in the local domain, either by having the local administrator create a new account, or by the resource owner sharing her password. Another approach is to install a gateway that accesses the resource with the local user's privilege but on behalf of the remote user. With the gateway the owner achieves her goal of sharing, but obscures the identity and authority of the actual client from the service that supplies the underlying resource.

Another way a user might share resources across administrative boundaries is by *delegating*¹ her authority with *restriction*. In the example, Alice may

¹We call *delegation* what Abadi et al. call *handoff*.

authorize Bob to perform some restricted set of actions on certain resources. Authority information flows across the administrative boundary: the delegation provides the resource server with sufficient information to reason about the client regardless of her membership in the local administrative domain. Indeed, the authorization mechanism has *no* inherent notion of administrative domain.

2.2 Spanning network scales

A second boundary that interferes with end-to-end authorization is network scale. Network scale affects an application's choice of hop-by-hop authorization protocol. For example, a strong encryption protocol is appropriate when crossing a wide-area network. Inside a firewall where routers are locally administered, some installations may base authority decisions on IP source addresses. On a local machine, we can often trust the OS kernel to correctly identify the participants in an interprocess communication.

Our unified approach separates policy from mechanism, creating two benefits. First, applications reason about policy using a toolkit with a narrow interface. The toolkit can transparently support multiple access mechanisms, and simply enable those that policy allows. Second, when an application does not support a desired mechanism, we can build a gateway that forwards requests from another mechanism while still passing end-to-end authorization information in a form the server understands and verifies. Ultimately, the high-level security analysis of a program is independent of mechanism, and reflects end-to-end trust relationships.

2.3 Spanning levels of abstraction

Another use for gateway programs is to introduce another level of abstraction over that provided by a lower-level resource server. A file system takes disk blocks and makes files; a calendar takes relational database records and makes events; a source-code repository takes files and makes configuration branches. Typically, an abstracting gateway controls the lower-level resource completely and exclusively, so that the gateway makes all access-control decisions. With end-to-end authorization, one can instead allow multiple mutually untrusting gateways to share a single lower-level resource.

For example, a system administrator might control the disk-block allocator. To grant Alice access to a specific file *X*, the sysadmin may allow Alice to speak for the file system regarding *X*, and allow the conjunction of Alice and the file system quoting Alice to speak for the disk blocks. In this configuration, the file system cannot access the lower-level

disk block resource without Alice's agreement (due to the conjunction), and Alice cannot meddle with arbitrary disk blocks without the file system agreeing that the requests are appropriate. The system helps us adhere to the principal of least privilege by encoding partial trust in the user and in the file system program. Furthermore, auditing any request for disk blocks provides end-to-end information indicating the involvement of both Alice and the file system program.

2.4 Spanning protocols

Commonly a gateway is installed between two systems simply to translate requests from one wire protocol to another. Like any gateway, these gateways often impede the flow of authorization information from client to server.

In our system, authorization information is encoded in a data structure that has both robust and efficient wire transfer encodings [18]. Thus the unified system is easily adapted for transfer over a variety of existing protocols. In this paper, we describe its implementation over HTTP and over Java Remote Method Invocation (RMI). Adapting more protocols, such as NFS and SMTP, to support unified authorization will result in wider applicability of end-to-end authorization.

The four boundaries described above turn up in real systems that accrete from smaller subsystems. Gateway software installed at each boundary maps requests from clients on one side of the boundary to requests for services on the other side. The system described herein allows us, at each boundary, to preserve the flow of authorization information alongside the flow of requests. By allowing gateways to defer authorization decisions to the final resource server when appropriate, and ensuring that resource servers have a full explanation for the authority of the requests they service, we provide applications with end-to-end authorization.

3 Unified authorization

Above, we motivate the use of a unified system to support end-to-end authorization, and allude to some of its features. In this section, we give an overview of the system we built, part of a project called Snowflake that facilitates naming and sharing across administrative boundaries.

The main idea behind our end-to-end authorization is a compact logic of authority. The logic is founded in a possible-worlds semantics that provides

intuition and guidance about possible extensions. Due to its length, the detailed semantics appears in a companion paper [11].

Logical assumptions represent statements that a principal believes based on some verification (outside the logic), such as the result of a digital signature verification. Principals combine assumptions and logical theorems to produce inherently auditable proofs of authority. Such proofs are not bearer capabilities but simply verifiable facts: while they prove that a given principal has authority, knowledge of the proof by an adversary does not bestow authority on the adversary. The primary form of statement is $B \stackrel{T}{\Rightarrow} A$, read “Bob speaks for Alice regarding the statements in set T .” The statement means that Alice agrees with Bob about any statement in T that Bob might make; the *speaks for* captures delegation, and the *regarding* captures restriction.

The logic stems from the Logic of Authentication due to Abadi, Burrows, Lampson, Plotkin, and Wobber [1, 13, 25]; as in their logic, ours can encode conjunction (multiple parties exercising joint authority) and quoting (one party claiming to speak on behalf of another). The logic is backed by a semantics that not only provides unambiguous meaning for every logical statement, but tells us how the system may and may not be safely extended.

The formalism suggests a natural implementation language that fits nicely with the Simple Public Key Infrastructure (SPKI) [9]. Our system generalizes SPKI by allowing other forms of principal, so that the same framework can be used for authorization on a single host using a trusted kernel, authorization within an administrative domain using a secret-key protocol, or authorization in the wide area using a public key protocol. We extended the SPKI framework rather than create our own to simplify potential interoperability with SPKI, to exploit SPKI’s unambiguous S-expression representation, and to build on existing implementations of SPKI in C and Java.

We present the implementation in three sections: the infrastructure of the system, the channels of communication we have supported, and some applications that exploit the authorization model. The applications culminate in a configuration that bridges each of the four boundaries described above.

Principals, statements and proofs are the language of our system. Section 4 describes each, and discusses our implementation. It also describes the Prover, a tool used by clients to generate proofs. Requests to be authorized are delivered over various kinds of channels, from fast local channels connected by a trusted kernel, to cryptographically-protected network connections. We discuss our implementation of authorization over channels in Section 5. In

Section 6, we describe the applications and services we have built that participate in and interoperate using the unified authorization system. We measure and analyze the costs of our approach in Section 7. Section 8 discusses related work, and we summarize in Section 9.

4 Infrastructure

The basic elements of the system are statements and principals. A statement is any assertion, such as “it would be good to read file X ,” or “Bob speaks for Alice,” or “Charlie says Alice speaks for Charlie.” A principal is any entity that can make a statement. Examples include the binary representation of a statement itself (that says only what it says), a cryptographic key (that says any message signed by the key), a secure channel (that says any message emanating from the channel), a program (that says its output), and a terminal (that says whatever the user types on it).

A proof of authority, like a proof of a mathematical theorem, is simply a collection of statements that together convince the reader of the veracity of the conclusion statement. Of course, in an authorization system, a proof is read by a program, not by a mathematician.

4.1 Statements

Snowflake’s implementation of sharing begins with the Java implementation of SPKI by Morcos [14]. It is a useful starting point because not only do we wish to preserve features of SPKI, but SPKI includes a precise and easily extensible specification of the representation of various abstractions. Furthermore, starting with a SPKI implementation offers an easier path to SPKI interoperability.

The restriction imposed on a delegation is specified using *authorization tags* from SPKI. Authorization tags concisely represent infinitely refinable sets, which makes them an attractive format for user-definable restrictions. We replaced Morcos’ minimal implementation of authorization tags with a complete one that performs arbitrary intersection operations [12, Chapter 6]. Our semantics paper explains how SPKI’s revocation mechanisms (lists and one-time revalidations) can be expressed as statements in our logic [11].

4.2 Principals

SPKI makes a distinction between principals and “subjects,” entities that can speak for others but can utter no statements directly, such as threshold

(conjunct) principals. Our formalism does not make that distinction. It also supports new compound principals, such as the quoting principal of Lampson et al. Therefore, we extended Morcos' Principal class to support SPKI threshold (conjunction) principals and Lampson's quoting principals. When a service reads a request from a communications channel, it associates the request with an appropriate principal object that represents the channel; this principal is the one that "says" the request. Because the channel itself is a principal, it may claim to quote some other principal; that assertion is noted by associating the channel with a Quoting principal object. The object's `quoter` field is the channel itself, and its `quotee` field is the (possibly compound) principal the channel claims to quote.

4.3 Proofs

We implemented a `Proof` class that represents a structured proof consisting of axioms and theorems of the logic and basic facts (delegations by principals). An instance of `Proof` describes the statement that it proves and can verify itself upon request. While `Proof` objects may be received from untrusted parties, their methods are loaded from a local code base, so that the results of verification are trustworthy. Servers receive from clients instances of the `Proof` class that show the client's authority to request service. Conversely, a server may send a `Proof` to a client to establish its authenticity, that is, to prove its authority to identify itself by some name or to provide some service the client expects.

Proofs can be transmitted as SPKI-style S-expressions or directly transferred between JVMs using Java serialization. No precision is lost in the latter case, since the basic internal structure of every proof component is a Java object corresponding to an S-expression.

SPKI's *sequence* objects also represent proofs of authority. SPKI sequences are poorly defined, but they are linear programs apparently intended to run on a simple verifier implemented as a stack machine. When certificates and opcodes are presented to the machine in the correct order, the machine arrives at the desired conclusion [8].

Transmitting proofs in a structured form rather than as SPKI sequences is attractive for three reasons. First, the structured proofs clearly exhibit their own meaning; to quote Abadi and Needham, "every message should say what it means" [2]. Second, the structured proof components map one-to-one to implementation objects that verify each component. The SPKI sequence verifier, in contrast, requires an external mapping to show that the state machine corresponds to correct application of the

formal logic. Third, it is simple to extract lemmas (subproofs) from structured proofs, allowing the prover to digest proofs into reusable components (Section 4.4).

The logic encodes expiration times as part of the restriction of a delegation, so that each proof need be verified only once. The step of matching a request to a proof automatically disregards expired conclusions, since a current request cannot match a conclusion with a restriction that it was valid only in the past. Figure 1 illustrates a proof. Since the structure of the proof is preserved, if the topmost statement should expire (perhaps because it depends on the short-lived statement $H_D \Rightarrow K_S$), the still-useful proof of $K_S \Rightarrow K_C \cdot N$ may be extracted and reused in future proofs.

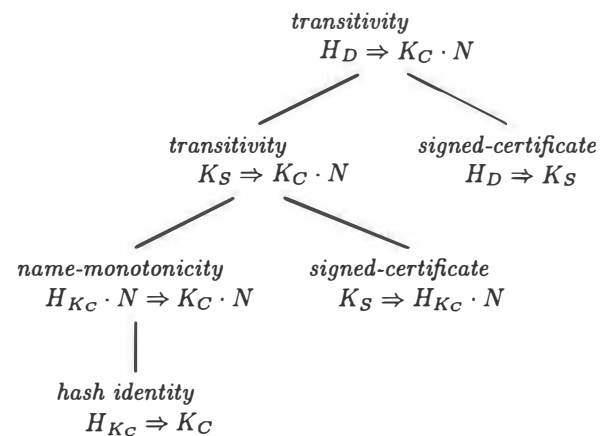


Figure 1: A structured proof. This proof shows that document D is the object client C associates with the name N . H_{K_C} is a hash of the client's key K_C , H_D a hash of the document, and K_S the server's key.

4.4 The prover

A `Prover` object helps Snowflake applications collect and create proofs. It has three tasks: it collects delegations, caches proofs, and constructs new delegations.

A user's application collects delegations from other users. Gateways collect delegations directly from client applications. Both sorts of applications use a `Prover` to maintain their collected delegations in a graph where nodes represent principals and edges represent a proof of authority from one principal to the next (see Figure 2). The `Prover` traverses the graph breadth first to find proofs of delegation required by the application. For example, if the `Prover` must prove that a channel K_{CH}

speaks for a server S , it works backwards from the node S to find the proof that $A \stackrel{V \cap X}{\Rightarrow} S$. A is *final*, meaning that the Prover can make statements as A ; therefore, Prover simply issues a delegation $K_{CH} \Rightarrow A$ to complete the proof.

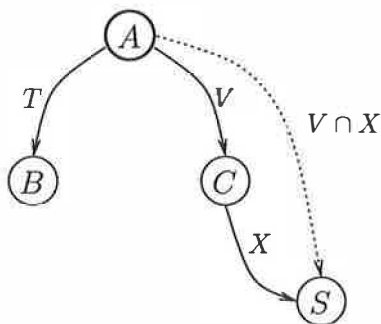


Figure 2: A look inside Alice's Prover. Each node represents a principal, and each edge a proof. For example, the edge from A to B represents the proof consisting of the single delegation $A \stackrel{T}{\Rightarrow} B$. The node A is distinguished because it is final: it represents a principal that the Prover can cause to say things.

When the Prover receives a delegation that is actually a proof involving several steps, the Prover “digests” the proof into its component parts for storage in the graph. Whenever it receives or computes a derived proof composed of smaller components, the Prover adds a shortcut edge (dotted line in Figure 2) to the graph to represent the proof. These shortcuts form a cache that eliminates most deep traversals of the graph.

When an application controls one or more principals (e.g., by holding the corresponding private key or capability), its Prover can store a closure (an object that knows the private key or how to exercise the capability) in its graph to represent the controlled principal. When desired, the Prover can not only find existing proofs, but complete new proofs by finding an existing chain of delegations from the controlled principal to the required issuer, then using the closure to delegate to the required subject restricted authority over the controlled principal.

Our simple Prover is incomplete, but it is suitable for most authorization tasks applications face. Abadi et al. note that solutions to the general access-control problem in the presence of both conjunction and quoting require exponential time [1, p.726]. Elieen gives a polynomial-time algorithm for discovering proofs in a graph with only SPKI certificates (no quoting principals) [7]. In the common case, we expect applications to collect authorization information in the course of resolving names, so that proofs

are built incrementally with graph traversals of constant depth.

5 Channels

With the infrastructure above in place, applications and services have the tools they need to generate, propagate, and analyze authority from the source of a request to its final resource server. The authorization information must be propagated from one program to the next through channels.

When a client makes a request of a server, the server needs some mechanism to ensure that the client really uttered the request. We implemented three such mechanisms: a secure network channel, a local channel vouched for by a trusted authority in the same (virtual) machine, and a signed request. We describe each and discuss how they are represented as principals in our unified system.

5.1 Secure channels

To implement a secure channel, we built a Java implementation of the `ssh` protocol that can interoperate with the Unix `sshd` service [26]. Then we built `JavaServerSocket` and `Socket` classes based on `ssh` that provide a secure connection. Either end of the connection can query its socket to discover the public key associated with the opposite end.²

We plugged our `ssh` sockets into RMI using socket factories. `Ssh` ensures that the channel is secure between some pair of public keys. To make that guarantee useful, we embody the channel as a principal. Consider the channel in Figure 3. To establish the channel, the server (principal P_S) uses public key K_1 and the client (P_C) key K_2 in the key exchange, and together they establish secret key K_{CH} as the symmetric session key.

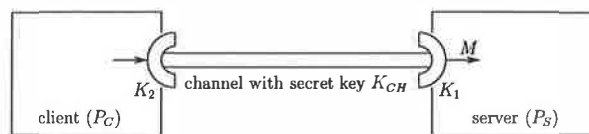


Figure 3: Treating a channel as a principal

Suppose a message M emerges from the channel at the server. In the language of the formalism,

²Why did we build an `ssh` implementation? Some have suggested that we use `SSL` over `RMI`, which is apparently now fairly practical. When we began this work, however, `RMI` did not have easily pluggable socket factories, and even once it did, the only open-source `SSL` implementation we could find did not operate well under `RMI`.

the `ssh` implementation promises that $M \Rightarrow K_{CH}$. The initial key exchange convinced the server that $K_{CH} \Rightarrow K_2$, and the client may explicitly establish that $K_2 \Rightarrow P_C$. Because $M \Rightarrow K_{CH} \Rightarrow K_2 \Rightarrow P_C$, the server concludes that $M \Rightarrow P_C$, that is, the message says what the client is thinking.

5.1.1 How channels work

Figure 4 illustrates our RMI/`ssh` channel in action. Initially, the server creates an instance of an RMI remote object **a**, defines the key K_S that controls it, and associates the object with an `SSHContext` that manages any incoming messages for the object **b**. The `SSHContext` is associated with the RMI listener socket **c** that will receive incoming requests for the object, and defines the public key (K_1) that will participate in `ssh` session establishment.

The client retrieves a stub **d** for the remote object from a name service it trusts. To exercise its authority on the object, the client first establishes its identity in thread scope. In a `try ... finally` block, it establishes its own `SSHContext` **e** and a `Prover` **f** that holds its private key K_C . Any method called in the run-time scope of the `try` block will inherit the established authority, but the authority will be canceled when control exits the block.

Then the client invokes a method m on the remote stub. The remote stub has been mechanically rewritten to wrap its remote invocations with calls to the `invoker` helper method **g**. The `invoker` method makes the usual RMI remote call through the remote reference **h**, and the reference creates an `ssh` socket **i** using the `SSHSocketFactory` specified in the stub. The `ssh` channel is established **j**, and each context learns the public key associated with the opposite end (K_1, K_2). The method call passes through the channel to the skeleton object on the server **k**, which forwards the call to the implementation object.

The programmer has prepended to each remote method implementation a call to the no-argument method `checkAuth()` **l**. This routine discovers from the local `SSHContext` the key K_2 associated with the channel that the request arrived on, and concludes K_2 says m . The server object was associated at creation with the key K_S , however, and `checkAuth()` does not know that K_2 speaks for K_S , so it throws an `SfNeedAuthorizationException`.

RMI passes the exception back through the channel, where the client's `invoker` method catches it. The `invoker` inspects the exception to discover the issuer K_S it must speak for and the minimum restriction set regarding which it must speak for that

issuer.³ The invoker queries the `Prover` **f** for a proof of the required authority; since the prover controls the client's private key K_C , it can construct a statement to delegate authority from K_C to K_2 . The exception carries a reference to a special remote `proofRecipient` object; the invoker calls a method on it to pass **m** the proof to the server. The `proofRecipient` object **n** stores the proof at the server, and returns to the client.

The invoker again sends the original invocation m through the remote reference, and the request travels the same path to `checkAuth` on the server. This time, the proof that $K_2 \stackrel{T}{\Rightarrow} K_S$ (via K_C) is available, `checkAuth()` returns without exception, and the remote object's implementation method runs to completion. Future calls encounter no exception as long as the proof at the server remains valid, and are only slowed by the layer of encryption protecting the integrity of the `ssh` channel.

The client programmer need only establish the client's authority at the top of a code block; inside that scope, the `Prover` and the invoker together handle the nitty-gritty of proof generation and authorization. In the idiom we adopt, the server programmer defines the object server key K_S and the mapping from method invocation to restriction set (T) for a server object, then prefixes each Remote method with calls to a generic `checkAuth()` that uses those definitions. We chose this approach because it would be simple to automate the injection of `checkAuth()` calls to insure that no Remote interface is left unprotected.

5.2 Local channels

Setting up a secure network channel is an expensive operation because it involves public-key operations to exchange keys. If a server trusts its host machine enough to run its software, it may as well trust the host to identify parties connected to local IPC channels. Within our Java environment, we treat the JVM and a few system classes as the trusted host, and bypass encryption when connecting to a server in the same JVM.

³In this example, the minimum restriction set $T = \{m\}$ contains the singleton request (method invocation) made by the invoker. When some more-sophisticated mapping is involved, where the server's minimum restriction set may reveal sensitive structure of the service, the server may reveal the set only incrementally. For example, its first challenge may tell the client how to prove authority to learn the "real" restriction set. The situation is analogous to `ls -l foo/bar` in Unix: it reveals the authority required by a client to access a resource `bar`, but only after the client has shown its authority to learn that information by logging in with a UID that has permission to read the directory `foo`.


```

HTTP/1.0 401 UNAUTHORIZED
Content-Type: text/html
MIME-Version: 1.0
Server: MortBay-Jetty-2.3.3
Date: Sat, 08 Apr 2000 15:18:47 GMT
WWW-Authenticate: SnowflakeProof
    Authorize-Client
Sf-ServiceIssuer: (hash md5
    |ehtQYd4EpQX0a/ON6Smesg=|)
Sf-MinimumTag: (tag
    (web (method GET)
        (service |Sm9uJ3MgUHHJvdGVjdGVpY2U=|)
        (resourcePath "")))
Connection: close

```

Figure 5: An HTTP authorization challenge message from a Snowflake server. It indicates the method, the required resource issuer, and the minimum restriction of a delegation that must be proven.

ferent security and performance trade-offs.

5.3.2 Authorization vs. authentication

The SPKI group argues that authorizing a request without authentication as an intermediate step reduces indirection and hence removes opportunities for attack [9]. When authentication is desired, one can use the logic to demand it. For example, one may delegate a resource to “authentication server’s Alice,” requiring Alice to authenticate herself to the server to invoke her authority over the resource. Alternatively, one can resolve the secure bindings that map keys to names after the fact to discover whose authority was invoked. How meaningful an authentication is depends on one’s philosophy about delegation control [11].

5.3.3 Server authorization

Often a client also wants to verify that it is communicating with the “right” server. The notion of “right” can be as simple as the server speaking for the client’s idea of a well-known name like `www.dartmouth.edu`, but in general the real question is still one of authorization: Does this server have the right to claim authority about Dartmouth’s course list? Does that server have authority to receive my e-mail?

We addressed a limited version of this problem with a second HTTP extension that enables a server to show the authenticity of a document using the authorization system. The server includes with document headers a proof that the hash of the document speaks for the server. The client completes the proof chain and determines whether the authentication is satisfactory.

5.3.4 Server implementation

We implement the server side of the signed-requests protocol as an abstract Java Servlet ProtectedServlet [15]. Concrete implementations extend ProtectedServlet with a method that maps a request to an issuer that controls the requested resource and to the minimum restriction set required to authorize the request. The concrete class also supplies the service implementation that maps a request to a response. When each request arrives, the ProtectedServlet ensures that appropriate authorization has been supplied, and if not, constructs and returns the “401 Unauthorized” response to the client.

Notice that the server identifies only a single principal that controls the resource, not an ACL. An ACL is a specific group of users authorized to access a resource; in our system, the client is responsible to know and exploit its group memberships as represented in delegations [11].

5.3.5 Client implementation

We realize our client as an HTTP proxy that enhances a browser with Snowflake authorization and server document-authentication services. Like any proxy, it forwards each HTTP request from the browser to a server. When a reply is “401 Unauthorized” and requires Snowflake authorization, the proxy uses its Prover to find a suitable proof, rewrites the request with an Authorization header, and retries the request.

The proxy provides an HTML user interface to its services at a virtual URL `http://security.localhost/`. Through this interface, the user can create a new private key pair, import principal identities and delegations, and delegate his authority to others. To delegate his authority, the user views a history of recently-visited pages, clicks the “delegate” link next to the page he wishes to share, and selects the recipient from a list of principals. The proxy generates an HTML snippet for the user to deliver to the recipient. A link inside the snippet names the destination page and carries both the delegation from the user as well as the proof the user needed to access the page. When the recipient follows the link, his own proxy imports the authorization information and redirects his browser to the named page.

6 Applications

We built three applications to demonstrate the Snowflake architecture for sharing.

6.1 Protected web server

The first application is simply a protected web file server that uses Snowflake's sharing architecture. One user establishes control over the file server by specifying the hash of his public key when starting up the server; he may delegate to others permission to read subtrees or individual files from the server using the mechanisms described above.

6.2 Protected database

The second application attaches Snowflake security to a relational email database. The original database server accepts insert, update, and select requests as RMI invocations on a Remote Database object, and returns the results of the query as serialized objects from the database. Adapting the application to Snowflake required only minimal changes. We modified the database instance constructor to use a `SshSocketFactory` so that all connections to the object use our `ssh` secure channels. Then, we prepended each implementation of a method in the remote interface with a call to the `checkAuth()` method. The database clients required only a modification to their initialization code to install an `SSHContext` and a `Prover`.

6.3 Quoting protocol gateway

The third application is a protocol gateway that provides an HTML over HTTP front-end to the email database. A database can be configured to allow certain principals access to certain data records. In the course of serving multiple users, the gateway can simultaneously access both Alice and Bob's email records. It is important that the gateway not misuse its authority and accidentally allow Bob to read Alice's email. The gateway programmer could try to prevent this mistake by checking access-control restrictions itself, but this approach duplicates the access control checks in the database, and increases the opportunity for error.

A better approach is to use quoting. The gateway's authority to access Alice's email in the database depends on the gateway intentionally quoting Alice in its requests. Therefore, as long as the gateway correctly quotes its clients in its requests on the database server, the correct access-control decision is made by the server.

A transaction begins when the client (C) sends an unauthorized request (R) to the gateway (G). The gateway queries the client for the identity the client wishes to use, and a delegation that the gateway speaks for the client to perform the task. The gateway attempts to access the database server (S),

but the RMI authorization fails because the gateway has no authority. The gateway sees an exception that indicates the required issuer S and restriction set (T). The gateway generates a "401 Unauthorized" Snowflake Authorization HTTP response, and in that response indicates it needs a proof that $G|? \stackrel{T}{\Rightarrow} S$. By $G|?$ the gateway means it needs a proof of authority that the gateway quoting the client speaks for the database. The client knows to substitute its identity for the "pseudo-principal" $?$; this shortcut saves a round-trip from the gateway to the client to discover the client's identity.

The client proxy now knows it needs to delegate its authority over the server to the principal "gateway quoting client," $G|C$. The client proxy generates the proof and submits it to the gateway along with a signed copy of its original request (showing $R \Rightarrow C$). The gateway digests the new proof and forwards the request to the database server. This time, the automatic RMI authorization protocol of Section 5.1.1 finds the proof in the gateway's `Prover`, and the database fulfills the request. The gateway builds an HTML interface from the database results for presentation to the user. Subsequent requests are accepted without so much fanfare, since the database server holds the appropriate proof of delegation.

The quoting gateway is a motivating application because it spans each of the four boundaries discussed in Section 2. Our gateway operates identically whether the client and the server are in the same administrative domain or different ones. It can be colocated with the server, in which case its RMI transactions automatically avoid encryption overhead by using the local channels of Section 5.2. The gateway constructs a view of an e-mail message from several rows and tables of a relational database, and so introduces a level of abstraction above the server resource. Finally, the gateway spans protocols by connecting an HTTP-speaking web browser with an RMI-speaking database server. Despite each of these boundaries, the gateway preserves the entire chain of authority that connects the client to the final server, enabling the server to make a fully-informed access-control decision.

6.3.1 Correctness and trust

The client trusts the gateway not to abuse the client's authority, and for some applications, the client may even trust the gateway to tell it how much authority the gateway needs to do its job. To establish that trust, the a client might first challenge the gateway to authenticate itself. If a gateway has received delegated authority from multiple clients (Alice and Bob), it must ensure that when it fulfills Bob's request it does not accidentally in-

voke Alice's authority. Where a conventional gateway would actually make access-control decisions to determine what Bob is allowed to do, our gateway only need be careful to correctly quote each client. It is therefore easier to verify that a quoting gateway is correct with respect to authorization.

In our system the notion of TCB is parameterized by the resource being protected. For example, the client software and hardware are part of the TCB for any resources the client is authorized to manipulate; when the client delegates a subset of those resources to the gateway, the gateway software and hardware become part of the TCB for that subset of resources. Although quoting helps us write the gateway application with greater confidence in its correctness, we cannot escape the fact that a compromised gateway still compromises the resources delegated to the gateway. Because the gateway is involved in the transfer of authority, authorization is not end-to-end in the pure sense of abstracting away intermediate steps. It is end-to-end, however, in the sense that authorization information now passes all the way from client to server, and the proof of authority verified by the server even includes evidence of the gateway principal's involvement.

7 Measurement

To better understand the costs of the Snowflake authorization model, and how they compare to costs of related systems, we timed the performance of our Snowflake-enhanced RMI implementation and our Snowflake-enhanced HTTP implementation. For comparison, we also timed standard RMI and standard HTTP servers with and without SSL support.

7.1 Experimental method

The values reported in this section are the parameters of linear regressions. In *setup cost and bandwidth* experiments, we vary the file length to separate copy cost from connection setup. In *setup and per-request* experiments, we vary the number of connections made after some slow setup operation to determine the amortizable part of the cost.

We made the measurements on 270 MHz Sun Ultra 5 hosts with 128 MB RAM, connected by a shared 10 Mbps Ethernet segment. The hosts run Solaris 2.7, Apache 1.3.12, OpenSSL 0.9.5, a locally-compiled Java JDK 1.2.2 with **green threads**, PureTLS 0.9b1, and Cryptix 3.1.1. We used 1024-bit RSA keys.

We ran each experiment ten times, discarding the first iteration so that caches are warm except where we intentionally measure setup costs. On each run,

we repeated an operation 10 to 1000 times, enough to amortize measurement overhead, and noted the total wall-clock time. When the nine runs had coefficient of variation greater than 0.1, we re-ran the experiment. We report values to two significant figures. The figures show values for single-machine experiments, where computation time, the dominant source of overhead, cannot hide under network latency. The raw data, complete tables of computed parameters, standard deviations and R^2 fitness coefficients are available [12, Chapter 12]. We computed 95% confidence intervals on the linear-regression parameters and found them vanishingly small.

7.2 RMI authorization with Snowflake

In this section, we quantify our implementation of Snowflake authorization over Java remote method invocation as described in Section 5.1. Figure 6 summarizes the overhead our prototype adds to RMI. The test operation is a Remote object that returns the contents of a file. Most of the overhead present in Snowflake is due to layering RMI over the `ssh` protocol. The extra work is the server's `checkAuth()` call, which retrieves the caller's public key, finds a cached proof for that subject, and sees that the proof has already been verified. The data-copy cost is unchanged compared to the `ssh` case.

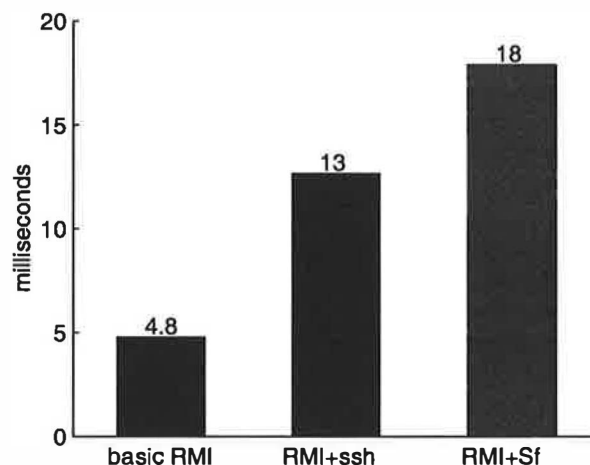


Figure 6: *The cost of introducing Snowflake authorization to RMI. A basic RMI call costs 4.8 ms. Securing the channel with `ssh` introduces significant overhead. Mapping the request into Snowflake and verifying the client's authority adds another 5 ms.*

It costs 470 ms to establish a new Snowflake-authorized RMI connection, reflecting the public-key

operation the client performs to delegate its authority to the channel. When the client caches the delegation but we make the server forget its copy after each use, we learn that the server spends 190 ms parsing and verifying the proof from the client.

7.3 HTTP authorization with Snowflake

In this section, we quantify our implementation of Snowflake authorization over the HTTP protocol as described in Section 5.3. As shown in Figure 7, the overhead of Java client and server code introduces a five-fold slowdown over an optimized C implementation of HTTP. Most of the rest of Snowflake's slowdown we have accounted for in the slow libraries described in Section 7.4.3.

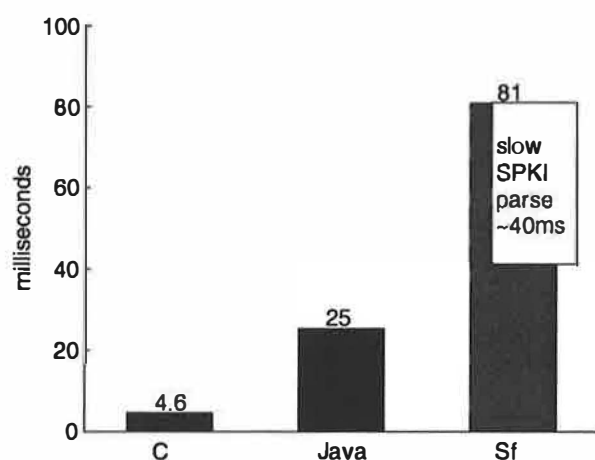


Figure 7: The cost of introducing Snowflake authorization to HTTP. A trivial C client accessing an Apache server takes 4.6 ms. Replacing the client and server with convenient but inefficient Java packages brings the baseline for HTTP to 25 ms. Most of Snowflake's overhead reflects the use of inefficient SPKI libraries, shown as an inset box.

The black bars in Figure 8 show our measurements of a Java SSL implementation, and the gray and white bars show the costs of the Snowflake authorization and document authentication protocols described in Section 5.3. Notice that when public-key encryption operations are involved, both protocols require hundreds of milliseconds. When caching connection information (Snowflake MAC protocol and identical requests versus a SSL request), they require tens of milliseconds. Snowflake's cached requests are a factor of two slower than SSL requests, due in part to differences in the protocol, and in part to the slow libraries discussed in Section 7.4.3.

Minimum cost of HTTP GET (C client and server)	5	5
Java+Jetty overhead for HTTP	20	20
Java SSL overhead	22	
S-expression parsing		~20
SPKI object unmarshalling		~20
Other Snowflake overhead (proof verification, SPKI object marshalling)		17
MAC costs (serialization, MD5 hash)		28
Total	47	110

Table 1: Breakdown of time spent in MAC authorization protocol. Units are milliseconds.

7.4 Observations

We hypothesize that the Snowflake authorization model is not prohibitively expensive. In fact, because it can subsume many hop-by-hop authorization models, it allows applications and users to make performance-security tradeoffs freely by selecting alternate hop-by-hop authorization protocols and plugging them into the same authorization framework.

Do our measurements support our hypothesis? Unfortunately, since our implementation is unoptimized and built on top of slow libraries, the numbers do not support our hypothesis unequivocally. By comparing them with baseline experiments, however, we believe we can make a strong case for the hypothesis. In the next two sections, we examine the two parts of our hypothesis. In Section 7.4.3, we argue that an optimized Snowflake promises to be competitive with existing hop-by-hop protocols.

7.4.1 Comparable operations

Snowflake-enhanced protocols are not inherently more expensive than other protocols with similar guarantees. The measurements displayed in Figure 8 indicate that Snowflake performs similar encryption steps as SSL. SSL spends about 400 ms starting up, as does Snowflake. SSL can complete a request over an established channel in about 50 ms. With our MAC optimization, a Snowflake request takes about 110 ms (see Table 1).

Both SSL and Snowflake engage in similar operations. SSL verifies message authenticity with symmetric-key decryption and a CRC; Snowflake does the same with an MD5 hash. Regardless of protocol, the server parses and processes the request and returns the reply. The SSL protocol checksums and encrypts the reply; Snowflake securely hashes

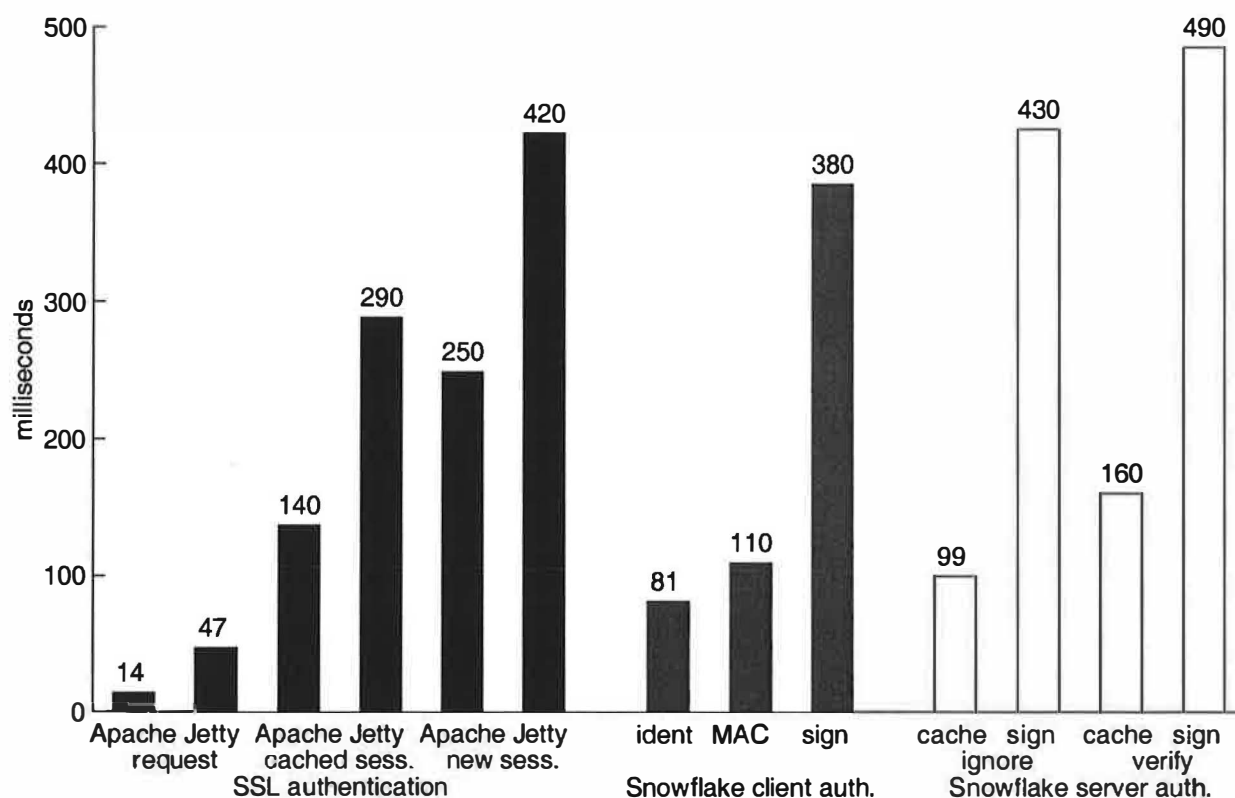


Figure 8: This graph displays the costs of standard SSL authentication (black bars) versus Snowflake client authentication (gray bars) and server document authentication (white bars).

the reply document. In both cases, the client uses a corresponding operation to verify the reply. Because the expensive cryptographic operations are comparable, one expects optimized implementations to perform comparably.

The additional sources of overhead in Snowflake are time spent walking the proof graph and memory consumed maintaining cached proofs. Our experiments do not explore that space in depth, but as we hint in Section 5.3.5, proofs are usually constructed incrementally while walking the name graph, an operation driven by the client user or application.

7.4.2 The performance–security tradeoff

By comparing our authorized-request protocol to SSL we somewhat compare apples and oranges, for the protocols make different performance–security tradeoffs. For example, our protocol does not verify the authenticity of the server’s reply header; since SSL provides integrity for the entire channel, a Snowflake–SSL protocol could as easily show the authenticity of all messages from the server.

In fact, part of the purpose of our system is

to enable such tradeoffs. With Snowflake, one is free to choose an established hop-by-hop protocol or to develop a new one. By stating in our logic the authorization promises the protocol makes, one can integrate the protocol into Snowflake’s end-to-end authorization model. Conceivably, new protocols can be dynamically integrated into existing Snowflake-aware applications; in other cases, a protocol-translating gateway can introduce the new protocol to the distributed system without hiding authorization information from the underlying application.

7.4.3 Slow libraries

Our formal measurements and informal tests indicate that a large fraction of Snowflake’s cost is needless overhead. Our baseline HTTP measurements indicate that using Java and the convenient Jetty web server incurs substantial overhead (250%). Furthermore, our SSL measurements indicate that the Java encryption library Cryptix imposes a substantial bandwidth overhead.

What surprised us most was the overhead of the

SPKI implementation on which we built Snowflake's objects. In informal tests, parsing a 2 KB S-expression from a string takes around 20 ms, and converting the resulting tree into typed Java objects takes another 20 ms. There is no reason a well-implemented library should spend milliseconds parsing short strings in a simple language; and 40+ ms delays such as these explain much of the difference between Snowflake's warm-connection performance and that of simple HTTP transactions (See Figure 7).

8 Related work

Our work is built primarily on the Logic of Authentication due to Abadi, Burrows, Lampson, Plotkin, and Wobber [1, 13, 25]. The Logic of Authentication introduced the notion of conjunct and quoting principals, and their applicability for modeling practical mechanisms such as channels and multiplexed gateways. We have preserved the generality and formality of the Logic of Authentication while introducing the crucial feature of restricted delegation. The structure of our implementation is similar to that of Taos, but we generally shift the burden of proof to the client so that the collection of access-control information happens in the course of name resolution as described in Section 4.4.

Sollins describes the restricted delegation problem as "cascaded authentication," and proposes as a solution a restricted delegation mechanism called *passports* [21] that provides for authorization of servers. Varadharajan et al. propose a more general mechanism that incorporates both symmetric and asymmetric encryption [23]. Neuman's *proxies* are tokens that express restricted delegation [17]. The Policy-Maker system has a notion of delegations with restrictions specified by arbitrary code [5]. As we mention in Section 3, SPKI has a notion of restricted delegation close to the one we use. Because the only principals in SPKI are public keys, it has high overhead for authorization on a single machine [9].

Sollins' passports, Neuman's proxies, Policy-Maker, and SPKI certificates are mechanisms with only informally-described semantics, and hence have no obvious and safe route to generalization. As we discuss in the companion paper [11], our formal semantics not only provides intuition for restricted delegation and end-to-end authorization, but it can advise us about the safety of possible extensions. Furthermore, it guides us in building a system with a minimal verification engine.

Appel and Felten's higher-order predicate logic is similarly inspired and applicable to SPKI [3]. Because our logic is a first-order propositional modal

logic, we can employ a conventional modal-logic semantics [11]. Our logic is also simpler; we factor implementation details out of the logic and leave only the structure of authorization. For example, concepts such as "digital signature" do not appear in our proof rules; instead, we integrate them by mapping a key to a logical principal, and asserting that a digital signature check validates the logical statement K says x .

Several single-machine operating systems have been built on the notion of restricted delegation; these are often called *capability-based* systems. Capabilities in KeyKOS, Eros, and Mach are unforgeable because the kernel manages them. A process delegates its authorization by asking the kernel to pass a capability, possibly with restriction, to another process [6, 20, 4]. Amoeba capabilities, in contrast, are secret random numbers, and may be transmitted as raw data [22, 16]. Amoeba must assume that a cluster is a secure network; we consider such a cluster a single administrative domain. Snowflake end-to-end authorization could integrate either sort of capability implementation as a fast, local authorization mechanism.

9 Summary and future work

We make a case for end-to-end authorization. Our proposal is based on a formal logic that models restricted delegations and hence models several existing hop-by-hop protocols. We describe the infrastructure of Snowflake, our implementation, including two hop-by-hop protocols and applications that exploit its end-to-end nature. Our end-to-end approach lets us connect systems with gateways that preserve authorization information, and by integrating multiple hop-by-hop mechanisms, it gives us freedom to easily trade off performance and security.

We would like to cross our work on end-to-end authorization with work on models of secrecy and information flow, to work toward an end-to-end model that can capture notions of who should *know* what. In such an architecture we imagine a gateway that operates with only partial access to the information it translates, passing from server to client encrypted content that it need not view to accomplish its task.

Acknowledgements

Thanks to Hany Farid for providing statistical intuition. We especially thank our reviewers and our shepherd Butler Lampson for all of their helpful comments and guidance.

References

- [1] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, Sept. 1993.
- [2] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, Jan. 1996.
- [3] A. W. Appel and E. W. Felten. Proof-carrying authentication. In *6th ACM Conference on Computer and Communications Security*, pages 52–62, Singapore, Nov. 1999. ACM Press.
- [4] D. Black, D. Golub, D. Julin, R. Rashid, R. Draves, R. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman. Microkernel operating system architecture and Mach. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 11–31, 1992.
- [5] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173, 1996.
- [6] A. Bomberger, A. Frantz, W. Frantz, A. Hardy, N. Hardy, C. Landau, and J. Shapiro. The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 95–112, 1992.
- [7] J.-E. Elien. Certificate discovery using SPKI/SDSI 2.0 certificates. Master's thesis, Massachusetts Institute of Technology, 1998.
- [8] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. M. Thomas, and T. Ylonen. Simple public key certificate. Internet draft `draft-ietf-spki-cert-structure-05.txt` (expired), Mar. 1998.
- [9] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. M. Thomas, and T. Ylonen. SPKI certificate theory, Oct. 1999. Internet RFC 2693.
- [10] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP authentication: Basic and digest access authentication, June 1999. Internet RFC 2617.
- [11] J. Howell and D. Kotz. A Formal Semantics for SPKI. In *Proceedings of the 6th European Symposium on Research in Computer Security*, Toulouse, France, Oct. 2000.
- [12] J. R. Howell. *Naming and sharing resources across administrative boundaries*. PhD thesis, Department of Computer Science, Dartmouth College, 2000.
- [13] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, Nov. 1992.
- [14] A. Morcos. A Java implementation of Simple Distributed Security Infrastructure. Master's thesis, Massachusetts Institute of Technology, May 1998.
- [15] K. Moss. *Java Servlets*. Computing McGraw-Hill, July 1998.
- [16] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: A distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, May 1990.
- [17] B. C. Neuman. Proxy-based authorization and accounting for distributed systems. In *Proceedings of the Thirteenth International Conference on Distributed Computing Systems*, pages 283–291, May 1993.
- [18] R. Rivest. S-Expressions. Internet draft `draft-rivest-sexp-00.txt` (expired), May 1997.
- [19] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, Nov. 1984.
- [20] J. Shapiro, J. Smith, and D. Farber. EROS: a fast capability system. *ACM Operating Systems Review*, 33(5):170–185, Dec. 1999.
- [21] K. R. Sollins. Cascaded authentication. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, pages 156–163, 1988.
- [22] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, Dec. 1990.
- [23] V. Varadharajan, P. Allen, and S. Black. An analysis of the proxy problem in distributed systems. In *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, pages 255–275, 1991.
- [24] V. Vydock and S. Kent. Security mechanisms in high-level network protocols. *Computing Surveys*, 15(2):135–171, 1983.
- [25] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, Feb. 1994.
- [26] T. Ylonen. The SSH (secure shell) remote login protocol. Internet draft `draft-ylonen-ssh-protocol-00.txt` (expired), May 1996.

Self-Securing Storage: Protecting Data in Compromised Systems

John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A.N. Soules

Gregory R. Ganger
Carnegie Mellon University

Abstract

Self-securing storage prevents intruders from undetectably tampering with or permanently deleting stored data. To accomplish this, self-securing storage devices internally audit all requests and keep old versions of data for a window of time, regardless of the commands received from potentially compromised host operating systems. Within the window, system administrators have this valuable information for intrusion diagnosis and recovery. Our implementation, called S4, combines log-structuring with journal-based metadata to minimize the performance costs of comprehensive versioning. Experiments show that self-securing storage devices can deliver performance that is comparable with conventional storage systems. In addition, analyses indicate that several weeks worth of all versions can reasonably be kept on state-of-the-art disks, especially when differencing and compression technologies are employed.

1 Introduction

Despite the best efforts of system designers and implementors, it has proven difficult to prevent computer security breaches. This fact is of growing importance as organizations find themselves increasingly dependent on wide-area networking (providing more potential sources of intrusions) and computer-maintained information (raising the significance of potential damage). A successful intruder can obtain the rights and identity of a legitimate user or administrator. With these rights, it is possible to disrupt the system by accessing, modifying, or destroying critical data.

Even after an intrusion has been detected and terminated, system administrators still face two difficult tasks: determining the damage caused by the intrusion and restoring the system to a safe state. Damage includes compromised secrets, creation of back doors and Trojan horses, and tainting of stored data. Detecting each of these is made difficult by crafty intruders who understand how to scrub audit logs and

disrupt automated tamper detection systems. System restoration involves identifying a clean backup (i.e., one created prior to the intrusion), reinitializing the system, and restoring information from the backup. Such restoration often requires a significant amount of time, reduces the availability of the original system, and frequently causes loss of data created between the safe backup and the intrusion.

Self-securing storage offers a partial solution to these problems by preventing intruders from undetectably tampering with or permanently deleting stored data. Since intruders can take the identity of real users and even the host OS, any resource controlled by the operating system is vulnerable, including the raw storage. Rather than acting as slaves to host OSes, self-securing storage devices view them, and their users, as questionable entities for which they work. These self-contained, self-controlled devices internally version all data and audit all requests for a guaranteed amount of time (e.g., a week or a month), thus providing system administrators time to detect intrusions. For intrusions detected within this window, all of the version and audit information is available for analysis and recovery. The critical difference between self-securing storage and host-controlled versioning (e.g., Elephant [29]) is that intruders can no longer bypass the versioning software by compromising complex OSes or their poorly-protected user accounts. Instead, intruders must compromise single-purpose devices that export only a simple storage interface, and in some configurations, they may have to compromise both.

This paper describes self-securing storage and our implementation of a self-securing storage server, called S4. A number of challenges arise when storage devices distrust their clients. Most importantly, it may be difficult to keep all versions of all data for an extended period of time, and it is not acceptable to trust the client to specify what is important to keep. Fortunately, storage densities increase faster than most computer characteristics (100%+ per annum in recent years). Analysis of recent workload studies [29, 34] suggests that it is possible to ver-

sion all data on modern 30–100GB drives for several weeks. Further, aggressive compression and cross-version differencing techniques can extend the intrusion detection window offered by self-securing storage devices. Other challenges include efficiently encoding the many metadata changes, achieving secure administrative control, and dealing with denial-of-service attacks.

The S4 system addresses these challenges with a new storage management structure. Specifically, S4 uses a log-structured object system for data versions and a novel journal-based structure for metadata versions. In addition to reducing space utilization, journal-based metadata simplifies background compaction and reorganization for blocks shared across many versions. Experiments with S4 show that the security and data survivability benefits of self-securing storage can be realized with reasonable performance. Specifically, the performance of S4-enhanced NFS is comparable to FreeBSD's NFS for both micro-benchmarks and application benchmarks. The fundamental costs associated with self-securing storage degrade performance by less than 13% relative to similar systems that provide no data protection guarantees.

The remainder of this paper is organized as follows. Section 2 discusses intrusion survival and recovery difficulties in greater detail. Section 3 describes how self-securing storage addresses these issues, identifies some challenges inherent to self-securing storage, and discusses design solutions for addressing them. Section 4 describes the implementation of S4. Section 5 evaluates the performance and capacity overheads of self-securing storage. Section 6 discusses a number of issues related to self-securing storage. Section 7 discusses related work. Section 8 summarizes this paper's contributions.

2 Intrusion Diagnosis and Recovery

Upon gaining access to a system, an intruder has several avenues of mischief. Most intruders attempt to destroy evidence of their presence by erasing or modifying system log files. Many intruders also install back doors in the system, allowing them to gain access at will in the future. They may also install other software, read and modify sensitive files, or use the system as a platform for launching additional attacks. Depending on the skill with which the intruders hide their presence, there will be some *detection latency* before the intrusion is discovered by an automated intrusion detection system (IDS) or by a suspicious user or administrator. During this

time, the intruders can continue their malicious activities while users continue to use the system, thus entangling legitimate changes with those of the intruders. Once an intrusion has been detected and discontinued, the system administrator is left with two difficult tasks: diagnosis and recovery.

Diagnosis is challenging because intruders can usually compromise the “administrator” account on most operating systems, giving them full control over all resources. In particular, this gives them the ability to manipulate everything stored on the system's disks, including audit logs, file modification times, and tamper detection utilities. Recovery is difficult because diagnosis is difficult and because user-convenience is an important issue. This section discusses intrusion diagnosis and recovery in greater detail, and the next section describes how self-securing storage addresses them.

2.1 Diagnosis

Intrusion diagnosis consists of three phases: detecting the intrusion, discovering what weaknesses were exploited (for future prevention), and determining what the intruder did. All are difficult when the intruder has free reign over storage and the OS.

Without the ability to protect storage from compromised operating systems, intrusion detection may be limited to alert users and system administrators noticing odd behavior. Examining the system logs is the most common approach to intrusion detection [7], but when intruders can manipulate the log files, such an approach is not useful. Some intrusion detection systems also look for changes to important system files [16]. Such systems are vulnerable to intruders that can change what the IDS thinks is a “safe” copy.

Determining how an intruder compromised the system is often impossible in conventional systems, because he will scrub the system logs. In addition, any *exploit tools* (utilities for compromising computer systems) that may have been stored on the target machine for use in multi-stage intrusions are usually deleted. The common “solutions” are to try to catch the intruder in the act or to hope that he forgot to delete his exploit tools.

The last step in diagnosing an intrusion is to discover what was accessed and modified by the intruder. This is difficult, because file access and modification times can be changed and system log files can be doctored. In addition, checksum databases are of limited use, since they are effective only for static files.

2.2 Recovery

Because it is usually not possible to diagnose an intruder's activities, full system recovery generally requires that the compromised machine be wiped clean and reinstalled from scratch. Prior to erasing the entire state of the system, users may insist that data, modified since the intrusion, be saved. The more effort that went into creating the changes, the more motivation there is to keep this data. Unfortunately, as the size and complexity of the data grows, the likelihood that tampering will go unnoticed increases. Foolproof assessment of the modified data is very difficult, and overlooked tampering may hide tainted information or a back door inserted by the intruder.

Upon restoring the OS and any applications on the system, the administrator must identify a backup that was made prior to the intrusion; the most recent backup may not be usable. After restoring data from a pre-intrusion backup, the legitimately modified data can be restored to the system, and users may resume using the system. This process often takes a considerable amount of time—time during which users are denied service.

3 Self-Securing Storage

Self-securing storage ensures information survival and auditing of all accesses by establishing a security perimeter around the storage device. Conventional storage devices are slaves to host operating systems, relying on them to protect users' data. A self-securing storage device operates as an independent entity, tasked with the responsibility of not only storing data, but protecting it. This shift of storage security functionality into the storage device's firmware allows data and audit information to be safeguarded in the presence of file server and client system intrusions. Even if the OSes of these systems are compromised and an intruder is able to issue commands directly to the self-securing storage device, the new security perimeter remains intact.

Behind the security perimeter, the storage device ensures data survival by keeping previous versions of the data. This *history pool* of old data versions, combined with the audit log of accesses, can be used to diagnose and recover from intrusions. This section discusses the benefits of self-securing storage and several core design issues that arise in realizing this type of device.

3.1 Enabling intrusion survival

Self-securing storage assists in intrusion recovery by allowing the administrator to view audit information and quickly restore modified or deleted files. The audit and version information also helps to diagnose intrusions and detect the propagation of maliciously modified data.

Self-securing storage simplifies detection of an intrusion since versioned system logs cannot be imperceptibly altered. In addition, modified system executables are easily noticed. Because of this, self-securing storage makes conventional tamper detection systems obsolete.

Since the administrator has the complete picture of the system's state, from intrusion until discovery, it is considerably easier to establish the method used to gain entry. For instance, the system logs would have normally been doctored, but by examining the versioned copies of the logs, the administrator can see any messages that were generated during the intrusion and later removed. In addition, any exploit tools temporarily stored on the system can be recovered.

Previous versions of system files, from before the intrusion, can be quickly and easily restored by resurrecting them from the history pool. This prevents the need for a complete re-installation of the operating system, and it does not rely on having a recent backup or up-to-date checksums (for tamper detection) of system files. After such restoration, critical data can be incrementally recovered from the history pool. Additionally, by utilizing the storage device's audit log, it is possible to assess which data might have been directly affected by the intruder.

The data protection that self-securing storage provides allows easy detection of modifications, selective recovery of tampered files, prevention of data loss due to out-of-date backups, and speedy recovery since data need not be loaded from an off-line archive.

3.2 Device security perimeter

The device's security model is what makes the ability to keep old versions more than just a user convenience. The security perimeter consists of self-contained software that exports only a simple storage interface to the outside world and verifies each command's integrity before processing it. In contrast, most file servers and client machines run a multitude of services that are susceptible to attack. Since the self-securing storage device is a single-

function device, the task of making it secure is much easier; compromising its firmware is analogous to breaking into an IDE or SCSI disk.

The actual protocol used to communicate with the storage device does not affect the data integrity that the new security perimeter provides. The choice of protocol does, however, affect the usefulness of the audit log in terms of the actions it can record and its correctness. For instance, the NFS protocol provides no authentication or integrity guarantees, therefore the audit log may not be able to accurately link a request with its originating client. Nonetheless, the principles of self-securing storage apply equally to “enhanced” disk drives, network-attached storage servers, and file servers.

For network-attached storage devices (as opposed to devices attached directly to a single host system), the new security perimeter becomes more useful if the device can verify each access as coming from both a valid user and a valid client. Such verification allows the device to enforce access control decisions and partially track propagation of tainted data. If clients and users are authenticated, accesses can be tracked to a single client machine, and the device’s audit log can yield the scope of direct damage from the intrusion of a given machine or user account.

3.3 History pool management

The old versions of objects kept by the device comprise the *history pool*. Every time an object is modified or deleted, the version that existed just prior to the modification becomes part of the history pool. Eventually an old version will age and have its space reclaimed. Because clients cannot be trusted to demarcate versions consisting of multiple modifications, a separate version should be kept for every modification. This is in contrast to versioning file systems that generally create new versions only when a file is closed.

A self-securing storage device guarantees a lower bound on the amount of time that a deprecated object remains in the history pool before it is reclaimed. During this window of time, the old version of the object can be completely restored by requesting that the drive *copy forward* the old version, thus making a new version. The guaranteed window of time during which an object can be restored is called the *detection window*. When determining the size of this window, the administrator must examine the trade-off between the detection latency provided by a large window and the extra disk space that is consumed by the proportionally larger history pool.

Although the capacity of disk drives is growing at a remarkable rate, it is still finite, which poses two problems:

1. Providing a reasonable detection window in exceptionally busy systems.
2. Dealing with malicious users that attempt to fill the history pool. (Note that space exhaustion attacks are not unique to self-securing storage. However, device-managed versioning makes conventional user quotas ineffective for limiting them.)

In a busy system, the amount of data written could make providing a reasonable detection window difficult. Fortunately, the analysis in Section 5.2 suggests that multi-week detection windows can be provided in many environments at a reasonable cost. Further, aggressive compression and differencing of old versions can significantly extend the detection window.

Deliberate attempts to overflow the history pool cannot be prevented by simply increasing the space available. As with most denial of service attacks, there is no perfect solution. There are three flawed approaches to addressing this type of abuse. The first is to have the device reclaim the space held by the oldest objects when the history pool is full. Unfortunately, this would allow an intruder to destroy information by causing its previous instances to be reclaimed from the overflowing history pool. The second flawed approach is to stop versioning objects when the history pool fills; although this will allow recovery of old data, system administrators would no longer be able to diagnose the actions of an intruder or differentiate them from subsequent legitimate changes. The third flawed approach is for the drive to deny any action that would require additional versions once the history pool fills; this would result in denial of service to all users (legitimate or not).

Our hybrid approach to this problem is to try to prevent the history pool from being filled by detecting probable abuses and throttling the source machine’s accesses. This allows human intervention before the system is forced to choose from the above poor alternatives. Selectively increasing latency and/or decreasing bandwidth allows well-behaved users to continue to use the system even while it is under attack. Experience will show how well this works in practice.

Since the history pool will be used for intrusion diagnosis and recovery, not just recovering from acci-

dental destruction of data, it is difficult to construct a safe algorithm that would save space in the history pool by pruning versions within the detection window. Almost any algorithm that selectively removes versions has the potential to be abused by an intruder to cover his tracks and to successfully destroy/modify information during a break-in.

3.4 History pool access control

The history pool contains a wealth of information about the system's recent activity. This makes accessing the history pool a sensitive operation, since it allows the resurrection of deleted and overwritten objects. This is a standard problem posed by versioning file systems, but is exacerbated by the inability to selectively delete versions.

There are two basic approaches that can be taken toward access control for the history pool. The first is to allow only a single administrative entity to have the power to view and restore items from the history pool. This could be useful in situations where the old data is considered to be highly sensitive. Having a single tightly-controlled key for accessing historical data decreases the likelihood of an intruder gaining access to it. Although this improves security, it prevents users from being able to recover from their own mistakes, thus consuming the administrator's time to restore users' files. The second approach is to allow users to recover their own old objects (in addition to the administrator). This provides the convenience of a user being able to recover their deleted data easily, but also allows an intruder, who obtains valid credentials for a given user, to recover that user's old file versions.

Our compromise is to allow users to selectively make this decision. By choice, a user could thus delete an object, version, or all versions from visibility by anyone other than the administrator, since permanent deletion of data via any other method than aging would be unsafe. This choice allows users to enjoy the benefits of versioning for presentations and source code, while preventing access to visible versions of embarrassing images or unsent e-mail drafts.

3.5 Administrative access

A method for secure administrative access is needed for the necessary but dangerous commands that a self-securing storage device must support. Such commands include setting the guaranteed detection window, erasing parts of the history pool, and accessing data that users have marked as "unrecoverable." Such administrative access can be securely

granted in a number of ways, including physical access (e.g., flipping a switch on the device) or well-protected cryptographic keys.

Administrative access is not necessary for users attempting to recover their own files from accidents. Users' accesses to the history pool should be handled with the same form of protection used for their normal accesses. This is acceptable for user activity, since all actions permitted for ordinary users can be audited and repaired.

3.6 Version and administration tools

Since self-securing storage devices store versions of raw data, users and administrators will need assistance in parsing the history pool. Tools for traversing the history must assist by bridging the gap between standard file interfaces and the raw versions that are stored by the device. By being aware of both the versioning system and formats of the data objects, utilities can present interfaces similar to that of Elephant [29], with "time-enhanced" versions of standard utilities such as `ls` and `cp`. This is accomplished by extending the read interfaces of the device to include an optional time parameter. When this parameter is specified, the drive returns data from the version of the object that was valid at the requested time.

In addition to providing a simple view of data objects in isolation, intrusion diagnosis tools can utilize the audit log to provide an estimate of damage. For instance, it is possible to see all files and directories that a client modified during the period of time that it was compromised. Further estimates of the propagation of data written by compromised clients are also possible, though imperfect. For example, diagnosis tools may be able to establish a link between objects based on the fact that one was read just before another was written. Such a link between a source file and its corresponding object file would be useful if a user determines that a source file had been tampered with; in this situation, the object file should also be restored or removed. Exploration of such tools will be an important area of future work.

4 S4 Implementation

S4 is a self-securing storage server that transparently maintains an efficient object-versioning system for its clients. It aims to perform comparably with current systems, while providing the benefits of self-securing storage and minimizing the corresponding space explosion.

RPC Type	Allows Time-Based Access	Description
Create	no	Create an object
Delete	no	Delete an object
Read	yes	Read data from an object
Write	no	Write data to an object
Append	no	Append data to the end of an object
Truncate	no	Truncate an object to a specified length
GetAttr	yes	Get the attributes of an object (S4-specific and opaque)
SetAttr	no	Set the opaque attributes of an object
GetACLByUser	yes	Get an ACL entry for an object given a specific UserID
GetACLByIndex	yes	Get an ACL entry for an object by its index in the object's ACL table
SetACL	no	Set an ACL entry for an object
PCreate	no	Create a partition (associate a name with an ObjectID)
PDelete	no	Delete a partition (remove a name/ObjectID association)
PList	yes	List the partitions
PMount	yes	Retrieve the ObjectID given its name
Sync	not applicable	Sync the entire cache to disk
Flush	not applicable	Removes all versions of all objects between two times
Flush0	not applicable	Removes all versions of an object between two times
SetWindow	not applicable	Adjusts the guaranteed detection window of the S4 device

Table 1: **S4 Remote Procedure Call List** – Operations that support time-based access accept a *time* in addition to the normal parameters; this *time* is used to find the appropriate version in the history pool. Note that all modifications create new versions without affecting the previous versions.

4.1 A self-securing object store

S4 is a network-attached object store with an interface similar to recent object-based disk proposals [9, 24]. This interface simplifies access control and internal performance enhancement relative to a standard block interface.

In S4, objects exist in a flat namespace managed by the “drive” (i.e., the object store). When objects are created, they are given a unique identifier (ObjectID) by the drive, which is used by the client for all future references to that object. Each object has an access control structure that specifies which entities (users and client machines) have permission to access the object. Objects also have metadata, file data, and opaque attribute space (for use by client file systems) associated with them.

To enable persistent mount points, a S4 drive supports “named objects.” The object names are an association of an arbitrary ASCII string with a particular ObjectID. The table of named objects is implemented as a special S4 object accessed through

dedicated partition manipulation RPC calls. This table is versioned in the same manner as other objects on the S4 drive.

4.1.1 S4 RPC interface

Table 1 lists the RPC commands supported by the S4 drive. The read-only commands (`read`, `getattr`, `getacl`, `plist`, and `pmount`) accept an optional *time* parameter. When the *time* is provided, the drive performs the read request on the version of the object that was “most current” at the time specified, provided that the user making the request has sufficient privileges.

The ACLs associated with objects have the traditional set of flags, with one addition—the **Recovery** flag. The **Recovery** flag determines whether or not a given user may read (recover) an object version from the history pool once it is overwritten or deleted. When this flag is clear, only the device administrator may read this object version once it is pushed into

the history pool. The Recovery flag allows users to decide the sensitivity of old versions on a file-by-file basis.

4.1.2 S4/NFS translation

Since one goal of self-securing storage is to provide an enhanced level of security and convenience on existing systems, the prototype minimizes changes to client systems. In keeping with this philosophy, the S4 drive is network-attached and an “S4 client” daemon serves as a user-level file system translator (Figure 1a). The S4 client translates requests from a file system on the target OS to S4-specific requests for objects. Because it runs as a user-level process, without operating system modifications, the S4 client should port to different systems easily.

The S4 client currently has the ability to translate NFS version 2 requests to S4 requests. The S4 client appears to the local workstation as a NFS server. This emulated NFS server is mounted via the loopback interface to allow only that workstation access to the S4 client. The client receives the NFS requests and translates them into S4 operations. NFSv2 was chosen over version 3 because its client is well-supported within Linux, and its lack of write caching allows the drive to maintain a detailed account of client actions.

Figure 1 shows two approaches to using the S4 client to serve NFS requests with the S4 drive. The first places the S4 client on the client system, as described previously, and uses the S4 drive as a network-attached storage device. The second incorporates the S4 client functionality into the server, as a NFS-to-S4 translator. This configuration acts as a S4-enhanced NFS server (Figure 1b) for normal file system activity, but recovery must still be accomplished through the S4 protocol since the NFS protocol has no notion of “time-based” access.

The implementation of the NFS file system overlays files and directories on top of S4 objects. Objects used as directories contain a list of ASCII filenames and their associated NFS file handles. Objects used as files and symlinks contain the corresponding data. The NFS attribute structure is maintained within the opaque attribute space of each object.

When the S4 client receives a NFS request, the NFS file handle (previously constructed by the S4 client) can be directly hashed into the ObjectID of the directory or file. The S4 client can then make requests directly to the drive for the desired data.

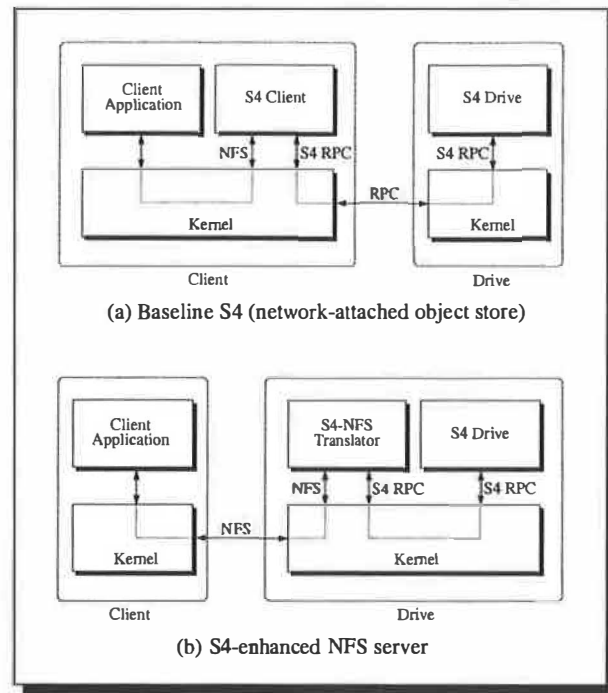


Figure 1: **Two S4 Configurations** – This figure shows two S4 configurations that provide self-securing storage via a NFS interface. (a) shows S4 as a network-attached object store with the S4 client daemon translating NFS requests to S4-specific RPCs. (b) shows a self-securing NFS server created by combining the NFS-to-S4 translation and the S4 drive.

To support NFSv2 semantics, the client sends an additional RPC to the drive to flush buffered writes to the disk at the end of each NFS operation that modifies the state of one or more objects. Since this RPC does not return until the synchronization is complete, NFSv2 semantics are supported even though the drive normally caches writes.

Because the client overlays a file system on top of the flat object namespace, some file system operations require several drive operations (and hence RPC calls). These sets of operations are analogous to the operations that file systems must perform on block-based devices. To minimize the number of RPC calls necessary, the S4 client aggressively maintains attribute and directory caches (for reads only). The drive also supports batching of `setattr`, `getattr`, and `sync` operations with `create`, `read`, `write`, and `append` operations.

4.2 S4 drive internals

The main goals for the S4 drive implementation are to avoid performance overhead and to minimize wasted space, while keeping all versions of all objects for a given period of time. Achieving these goals re-

quires a combination of known and novel techniques for organizing on-disk data.

4.2.1 Log-structuring for efficient writes

Since data within the history pool cannot be overwritten, the S4 drive uses a log structure similar to LFS [27]. This structure allows multiple data and metadata updates to be clustered into fewer, larger writes. Importantly, it also obviates any need to move previous versions before writing.

In order to prune old versions and reclaim unused segments, S4 includes a background cleaner. While the goal of this cleaner is similar to that of the LFS cleaner, the design must be slightly different. Specifically, deprecated objects cannot be reclaimed unless they have also aged out of the history pool. Therefore, the S4 cleaner searches through the object map for objects with an oldest time greater than the detection window. Once a suitable object is found, the cleaner permanently frees all data and metadata older than the window. If this clears all of the resources within a segment, the segment can be marked as free and used as a fresh segment for foreground activity.

4.2.2 Journal-based metadata

To efficiently keep all versions of object metadata, S4 uses *journal-based metadata*, which replaces most instances of metadata with compact journal entries.

Because clients are not trusted to notify S4 when objects are closed, every update creates a new version and thus new metadata. For example, when data pointed to by indirect blocks is modified, the indirect blocks must be versioned as well. In a conventional versioning system, a single update to a triple-indirect block could require four new blocks as well as a new inode. Early experiments with this type of versioning system showed that modifying a large file could cause up to a 4x growth in disk usage. Conventional versioning file systems avoid this performance problem by only creating new versions when a file is closed.

In order to significantly reduce these problems, S4 encodes metadata changes in a journal that is maintained for the duration of the detection window. By persistently keeping journal entries of all metadata changes, metadata writes can be safely delayed and coalesced, since individual inode and indirect block versions can be recreated from the journal. To avoid

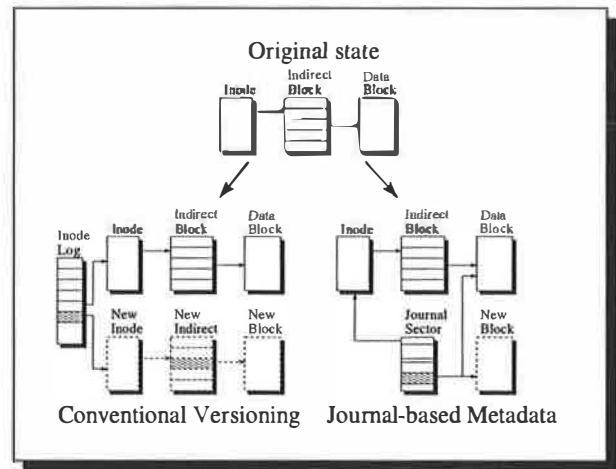


Figure 2: **Efficiency of Metadata Versioning** – The above figure compares metadata management in a conventional versioning system to S4’s journal-based metadata approach. When writing to an indirect block, a conventional versioning system allocates a new data block, a new indirect block, and a new inode. Also, the identity of the new inode must be recorded (e.g., in an Elephant-like inode log). With journal-based metadata, a single journal entry suffices, pointing to both the new and old data blocks.

rebuilding an object’s current state from the journal during normal operation, an object’s metadata is checkpointed to a log segment before being evicted from the cache. Unlike conventional journaling, such checkpointing does not prune journal space; only aging may prune space. Figure 2 depicts the difference in disk space usage between journal-based metadata and conventional versioning when writing data to an indirect data block.

In addition to the entries needed to describe metadata changes, a *checkpoint* entry is needed. This *checkpoint* entry denotes writing a consistent copy of all of an object’s metadata to disk. It is necessary to have at least one checkpoint of an object’s metadata on disk at all times, since this is the starting point for all time-based and crash recovery recreations.

Storing an object’s changes within the log is done using *journal sectors*. Each journal sector contains the packed journal entries that refer to a single object’s changes made within that segment. The sectors are identified by segment summary information. Journal sectors are chained together backward in time to allow for version reconstruction.

Journal-based metadata can also simplify cross-version differential compression [3]. Since the blocks changed between versions are noted within each entry, it is easy to find the blocks that should be com-

pared. Once the differencing is complete, the old blocks can be discarded, and the difference left in its place. For subsequent reads of old versions, the data for each block must be recreated as the entries are traversed. Cross-version differencing of old data will often be effective in reducing the amount of space used by old versions. Adding differencing technology into the S4 cleaner is an area of future work.

4.2.3 Audit log

In addition to maintaining previous object versions, S4 maintains an append-only audit log of all requests. This log is implemented as a reserved object within the drive that cannot be modified except by the drive itself. However, it can be read via RPC operations. The data written to the audit log includes command arguments as well as the originating client and user. All RPC operations (read, write, and administrative) are logged. Since the audit log may only be written by the drive front end, it need not be versioned, thus increasing space efficiency and decreasing performance costs.

5 Evaluation of self-securing storage

This section evaluates the feasibility of self-securing storage. Experiments with S4 indicate that comprehensive versioning and auditing can be performed without a significant performance impact. Also, estimates of capacity growth, based on reported workload characterizations, indicate that history windows of several weeks can easily be supported in several real environments.

5.1 Performance

The main performance goal for S4 is to be comparable to other networked file systems while offering enhanced security features. This section demonstrates that this goal is achieved and also explores the overheads specifically associated with self-securing storage features.

5.1.1 Experimental Setup

The four systems used in the experiments had the following configurations: (1) a S4 drive running on RedHat 6.1 Linux communicating with a Linux client over S4 RPC through the S4 client module (Figure 1a), (2) a S4-enhanced NFS server running

on RedHat 6.1 Linux communicating with a Linux client over NFS (Figure 1b), (3) a FreeBSD 4.0 server communicating with a Linux client over NFS, and (4) a RedHat 6.1 Linux server communicating with a Linux client over NFS. Since Linux NFS does not comply with the NFSv2 semantics of committing data to stable storage before operation completion, the Linux server's file system was mounted synchronously to approximate NFS semantics. In all cases, NFS was configured to use 4KB read/write transfer sizes, the only option supported by Linux. The FreeBSD NFS configuration exports a BSD FFS file system, while the Linux NFS configuration exports an ext2 file system. All experiments were run five times and have a standard deviation of less than 3% of the mean. The S4 drives were configured with a 128MB buffer cache and a 32MB object cache. The Linux and FreeBSD NFS servers' caches could grow to fill local memory (512MB).

In all experiments, the client system has a 550MHz Pentium III, 128MB RAM, and a 3Com 3C905B 100Mb network adapter. The servers have a 600MHz Pentium III, 512MB RAM, a 9GB 10,000RPM Ultra2 SCSI Seagate Cheetah drive, an Adaptec AIC-7896/7 Ultra2 SCSI controller, and an Intel Ether-Express Pro100 100Mb network adapter. The client and server are on the same subnet and are connected by a 100Mb network switch. All versions of Linux use an unmodified 2.2.14 kernel, and the BSD system uses a stock FreeBSD 4.0 installation.

To evaluate performance for common workloads, results from two application benchmarks are presented: the PostMark benchmark [14] and the SSH-build benchmark [36]. These benchmarks crudely represent Internet server and software development workloads, respectively.

PostMark was designed to measure the performance of a file system used for electronic mail, netnews, and web based services. It creates a large number of small randomly-sized files (between 512B and 9KB) and performs a specified number of transactions on them. Each transaction consists of two sub-transactions, with one being a create or delete and the other being a read or append. The default configuration used for the experiments consists of 20,000 transactions on 5,000 files, and the biases for transaction type are equal.

The SSH-build benchmark was constructed as a replacement for the Andrew file system benchmark [12]. It consists of 3 phases: The unpack phase, which unpacks the compressed tar archive of SSH v1.2.27 (approximately 1MB in size before decom-

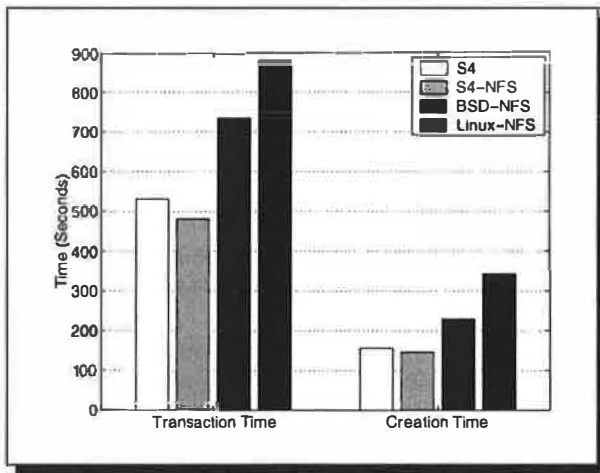


Figure 3: PostMark Benchmark

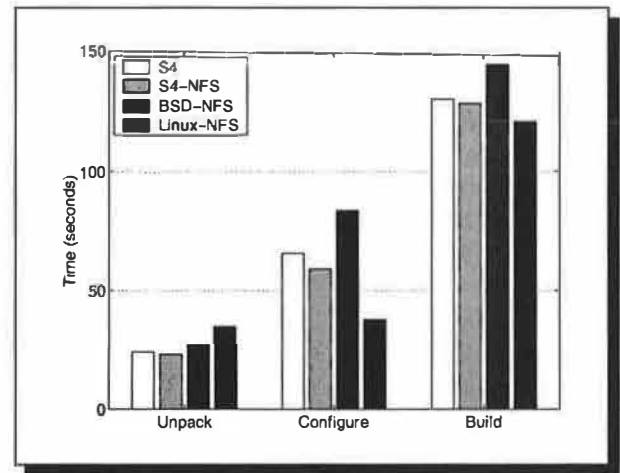


Figure 4: SSH-build Benchmark

pression), stresses metadata operations on files of varying sizes. The configure phase consists of the automatic generation of header files and Makefiles, which involves building various small programs that check the existing system configuration. The build phase compiles, links, and removes temporary files. This last phase is the most CPU intensive, but it also generates a large number of object files and a few executables.

5.1.2 Comparison of the servers

To gauge the overall performance of S4, the four systems described earlier were compared. As hoped, S4 performs comparably to the existing NFS servers.

Figure 3 shows the results of the PostMark benchmark. The times for both the creation (time to create the initial 5000 files) and transaction phases of PostMark are shown for each system. The S4 systems' performance is similar to both BSD and Linux NFS performance, doing slightly better due to their log structured layout.

The times of SSH-build's three phases are shown in Figure 4. Performance is similar across the S4 and BSD configurations. The superior performance of the Linux NFS server in the configure stage is due to a much lower number of write I/Os than in the BSD and S4 servers, apparently due to a flaw in the synchronous mount option under Linux.

5.1.3 Overhead of the S4 cleaner

In addition to the more visible process of creating new versions, S4 must eventually garbage collect data that has expired from the history pool. This garbage collection comes at a cost. The potential overhead of the cleaner was measured by running the PostMark benchmark with 50,000 transactions on increasingly large sets of initial files. For each set of initial files, the benchmark was run once with the cleaner disabled and once with the cleaner competing with foreground activity.

The results shown in Figure 5 represent PostMark running with the initial set of files filling between 2% and 90% of a 2GB disk. As expected, when the working set increases, performance of the normal S4 system degrades due to increasingly poor cache and disk locality. The sharp drop in the graph from 2% to 10% is caused by the fact that the set of files and data expands beyond the bounds of the drive's cache.

Although the S4 cleaner is slightly different, it was expected to behave similarly to a standard LFS cleaner, which has up to an approximate 34% decrease in performance [30]. The S4 cleaner is slightly more intrusive, degrading performance by approximately 50% in the worst case. The greater degradation is attributed mainly to the additional reads necessary when cleaning objects rather than segments. In addition, the S4 cleaner has not been tuned and does not include known techniques for reducing cleaner performance problems [21].

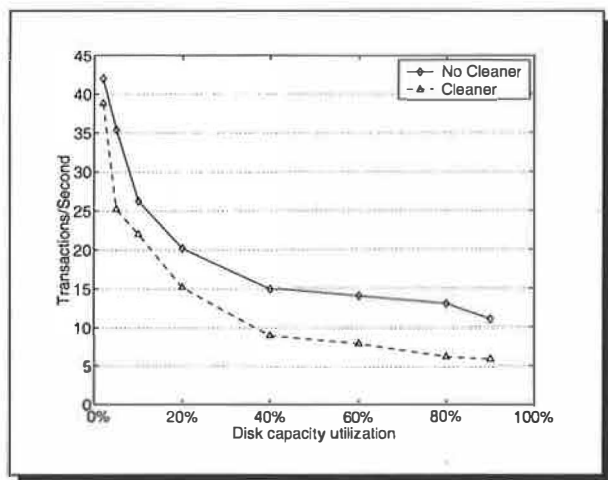


Figure 5: Overhead of foreground cleaning in S4 – This figure shows the transaction performance of S4 running the PostMark benchmark with varying capacity utilizations. The solid line shows system performance on a system without cleaning. The dashed line shows system performance in the presence of continuous foreground cleaner activity.

5.1.4 Overhead of the S4 audit log

In addition to versioning, self-securing storage devices keep an audit log of all connections and commands sent to the drive. Recording this audit log of events has some cost. In the worst case, all data written to the disk belongs to the audit log. In this case, one disk write is expected approximately every 750 operations. In the best case, large writes, the audit log overhead is almost non-existent, since the writes of the audit log blocks are hidden in the segment writes of the requests. For the macro-benchmarks, the performance penalty ranged between 1% and 3%.

For a more focused view of this overhead, a set of micro-benchmarks were run with audit logging enabled and disabled. The micro-benchmarks proceed in three phases: creation of 10,000 1KB files (split across 10 directories), reads of the newly created files in creation order, and deletion of the files in creation order.

Figure 6 shows the results. The create and delete phases exhibit a 2.8% and 2.9% decrease in performance, respectively, and the read phase exhibits a 7.2% decrease in performance. Read performance suffers a larger penalty because the audit log blocks become interwoven with the data blocks in the create phase. This reduces the number of files packed into each segment, which in turn increases the number of segment reads required.

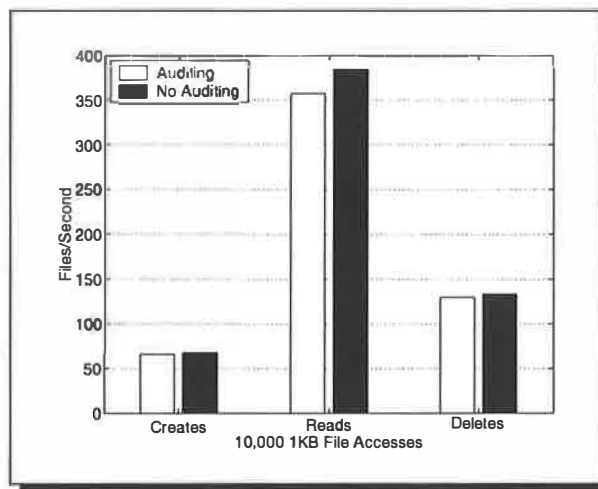


Figure 6: Auditing Overhead in S4 – This figure shows the impact on small file performance caused by auditing incoming client requests.

5.1.5 Fundamental performance costs

There are three fundamental performance costs of self-securing storage: versioning, auditing, and garbage collection. Versioning can be achieved at virtually no cost by combining journal-based metadata with the LFS structure. Auditing creates a small performance penalty of 1% to 3%, according to application benchmarks. The final performance cost, garbage collection, is more difficult to quantify. The extra overhead of S4 cleaning in comparison to standard LFS cleaning comes mainly from the difference in utilized space due to the history pool.

The worst-case performance penalty for garbage collection in S4 can be estimated by comparing the cleaning overhead at two space utilizations: the space utilized by the active set of objects and the space utilized by the active set combined with the history pool. For example, assume that the active set utilizes 60% of the drive's space and the history pool another 20%. For PostMark, the cleaning overhead is the difference between cleaning performance and standard performance seen at a given space utilization in Figure 5. For 60% utilization, the cleaning overhead is 43%. For 80% utilization, it is 53%. Thus, in this example, the extra cleaning overhead caused by keeping the history pool is 10%.

There are several possibilities for reducing cleaner overhead for all space utilizations. With expected detection windows ranging into the hundreds of days, it is likely that the history pool can be extended until such a time that the drive becomes idle. During idle time, the cleaner can run with no observ-

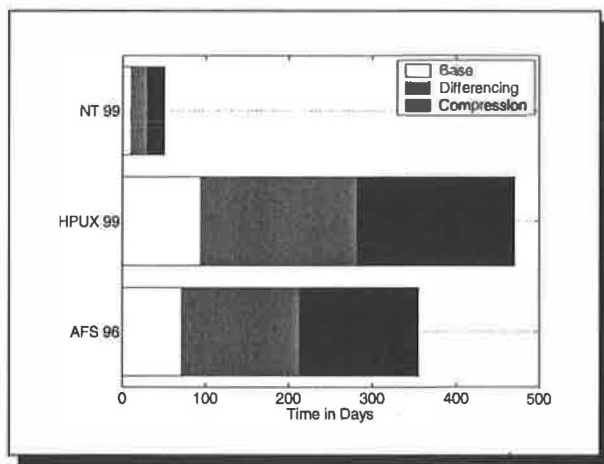


Figure 7: Projected Detection Window – The expected detection window that could be provided by utilizing 10GB of a modern disk drive. This conservative history pool would consume only 20% of a 50GB disk's total capacity. The base-line number represents the projected number of days worth of history information that can be maintained within this 10GB of space. The gray regions show the projected increase that cross-version differencing would provide. The black regions show the further increase expected from using compression in addition to differencing.

able overhead [2]. Also, recent research into technologies such as freeblock scheduling offer standard LFS cleaning at almost no cost [18]. This technique could be extended for cleaning in S4.

5.2 Capacity Requirements

To evaluate the size of the detection window that can be provided, three recent workload studies were examined. Figure 7 shows the results of approximations based on worst-case write behavior. Spasojevic and Satyanarayanan's AFS trace study [32] reports approximately 143MB per day of write traffic per file server. The AFS study was conducted using 70 servers (consisting of 32,000 cells) distributed across the wide area, containing a total of 200GB of data. Based on this study, using just 20% of a modern 50GB disk would yield over 70 days of history data. Even if the writes consume 1GB per day per server, as was seen by Vogels' Windows NT file usage study [34], 10 days worth of history data can be provided. The NT study consisted of 45 machines split into personal, shared, and administrative domains running workloads of scientific processing, development, and other administrative tasks. Santry, et al. [29] report a write data rate of 110MB per day. In this case, over 90 days of data could be kept. Their environment consisted of a single file system holding 15GB of data that was being used

by a dozen researchers for development.

Much work has been done in evaluating the efficiency of differencing and compression [3, 4, 5]. To briefly explore the potential benefits for S4, its code base was retrieved from the CVS repository at a single point each day for a week. After compiling the code, both differencing and differencing with compression were applied between each tree and its direct neighbor in time using Xdelta [19, 20]. After applying differencing, the space efficiency increased by 200%. Applying compression added an additional 200% for a total space efficiency of 500%. These results are in line with previous work. Applying these estimates to the above workloads indicates that a 10GB history pool can provide a detection window of between 50 and 470 days.

6 Discussion

This section discusses several important implications of self-securing storage.

Selective versioning: There are data that users would prefer not to have backed up at all. The common approach to this is to store them in directories known to be skipped by the backup system. Since one of the goals of S4 is to allow recovery of exploit tools, it does not support designating objects as non-versioned. A system may be configured with non-S4 partitions to support selective versioning. While this would provide a way to prevent versioning of temporary files and other non-critical data, it would also create a location where an intruder could temporarily store exploit tools without fear that they will be recovered.

Versioning vs. snapshots: Self-securing storage can be implemented with frequent copy-on-write snapshots [11, 12, 17] instead of versioning, so long as snapshots are kept for the full detection window. Although the audit log can still provide a record of what blocks are changed, snapshots often will not allow administrators to recover short-lived files (e.g., exploit tools) or intermediate versions (e.g., system log file updates). Also, legitimate changes are only guaranteed to survive malicious activity if they survive to the next snapshot time. Of course, the potential scope of such problems can be reduced by shrinking the time between snapshots. The comprehensive versioning promoted in this paper represents the natural end-point of such shrinking—every modification creates a new snapshot.

Versioning file systems vs. self-securing storage: Versioning file systems excel at providing users

with a safety net for recovery from accidents. They maintain old file versions long after they would be reclaimed by the S4 system, but they provide little additional system security. This is because they rely on the host's OS for security and aggressively prune apparently insignificant versions. By combining self-securing storage with long-term landmark versioning [28], recovery from users' accidents could be enhanced while also maintaining the benefits of intrusion survival.

Self-securing storage for databases: Most databases log all changes in order to protect internal consistency in the face of system crashes. Some institutions also retain these logs for long-term auditing purposes. All information needed to understand and recover from malicious behavior can be kept, in database-specific form, in these logs. Self-securing storage can increase the post-intrusion recoverability of database systems in two ways: (1) by preventing undetectable tampering with stored log records, and (2) by preventing undetectable changes to data that bypass the log. After an intrusion, self-securing storage allows a database system to verify its log's integrity and confirm that all changes are correctly reflected in the log—the database system can then safely use its log for subsequent recovery.

Client-side cache effects: In order to improve efficiency, most client systems use caches to minimize storage latencies. This is at odds with the desire to have storage devices audit users' accesses and capture exploit tools. Client-side read caches hide data dependency information that would otherwise be available to the drive in the form of reads followed quickly by writes. However, this information could be provided by client systems as (questionable) hints during writes. Write caches cause a more serious problem when files are created then quickly deleted, thus never being sent to the drive. This could cause difficulties with capturing exploit tools, since they may never be written to the drive. Although client cache effects may obscure some of the activity in the client system, data that are stored on a self-securing storage device are still completely protected.

Object-based vs. block-based storage: Implementing a self-securing storage device with a block interface adds several difficulties. Since objects are designed to contain one data item (file or directory), enforcing access control at this level is much more manageable than attempting to assign permissions on a per-block basis. In addition, maintaining versions of objects as a whole, rather than having to collect and correlate individual blocks, simplifies recovery tools and internal reorganization mechanisms.

Multi-device coordination: Multi-device coordination is necessary for operations such as striping data or implementing RAID across multiple self-securing disks or file servers. In addition to the coordination necessary to ensure that multiple copies of data are synchronized, recovery operations must also coordinate old versions. On the other hand, clusters of self-securing storage devices could maintain a single history pool and balance the load of versioning objects. Note that a self-securing storage device containing several disks (e.g., a self-securing disk array) does not have these issues. Additionally, it has the ability to keep old versions and current data on separate disks.

7 Related Work

Self-securing storage and S4 build on many ideas from previous work. Perhaps the clearest example is versioning: many versioned file systems have helped their users to recover from mistakes [22, 10]. Santry, et al., provide a good discussion of techniques for traversing versions and deciding what to retain [29]. S4's history pool corresponds to Elephant's "keep all" policy (during its detection window), and it uses Elephant's time-based access. The primary advantage of S4 over such systems is that it has been partitioned from client operating systems. While this creates another layer of abstraction, it adds to the survivability of the storage.

A self-securing disk drive would be another instance of many recent "smart disk" systems [1, 8, 15, 26, 35]. All of these exploit the increasing computation power of such devices. Some also put these devices on networks and exploit an object-based interface. There is now an ANSI X3T10 (SCSI) working group looking to create a new standard for object-based storage devices. The S4 interface is similar to these.

The standard method of intrusion recovery is to keep a periodic backup of files on trusted storage. Several file systems simplify this process by allowing a snapshot to be taken of a file system [11, 12, 17]. This snapshot can then be backed-up with standard file system tools. Spirallog [13] uses a log-structured file system to allow for backups to be made during system operation by simply recording the entire log to tertiary storage. While these systems are effective in preventing the loss of long-existing critical data, the window of time in which data can be destroyed or tampered with is much larger than in S4—often 24 hours or more. Also, these systems are generally reliant upon a system administrator for operation, with a corresponding increase in cost and potential

for human error. In addition, intrusion diagnosis is extremely difficult in such systems. Permanent file storage [25] provides an unlimited set of puncture-proof backups over time. These systems are unlikely to become the first-line of storage because of lengthy access times.

S4 borrows on-disk data structures from several systems. Unlike Elephant's FFS-like layout [23], the disk layout of S4 more closely resembles that of a log structured file system [27]. Many file systems use journaling to improve performance while maintaining disk consistency [6, 31, 33]. However, these systems delete the journal information once checkpoints ensure that the corresponding blocks are all on disk. S4's journal-based metadata persistently stores metadata versions in a space-efficient manner.

8 Conclusions

Self-securing storage ensures data and audit log survival in the presence of successful intrusions and even compromised host operating systems. Experiments with the S4 prototype show that self-securing storage devices can achieve performance that is comparable to existing storage appliances. In addition, analysis of recent workload studies suggest that complete version histories can be kept for several weeks on state-of-the-art disk drives.

Acknowledgments

We thank Brian Bershad, David Petrou, Garth Gibson, Andy Klosterman, Alistair Veitch, Jay Wylie, and the anonymous reviewers for helping us refine this paper. We thank the members and companies of the Parallel Data Consortium (including CLARiON, EMC, HP, Hitachi, Infineon, Intel, LSI Logic, MTI, Novell, PANASAS, Procom, Quantum, Seagate, Sun, Veritas, and 3Com) for their interest, insights, and support. We also thank IBM Corporation for supporting our research efforts. This work is partially funded by the National Science Foundation via CMU's Data Storage Systems Center and by DARPA/ISO's Intrusion Tolerant Systems program (Air Force contract number F30602-99-2-0539-AFRL). Craig Soules is supported by a USENIX scholarship.

References

- [1] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: programming model, algorithms and evaluation. *Architectural Support for Programming Languages and Operating Systems* (San Jose, California), pages 81–91. ACM, 3–7 October 1998.
- [2] Trevor Blackwell, Jeffrey Harris, and Margo Seltzer. Heuristic cleaning algorithms in log-structured file systems. *Annual USENIX Technical Conference* (New Orleans), pages 277–288. Usenix Association, 16–20 January 1995.
- [3] Randal C. Burns. *Differential compression: a generalized solution for binary files*. Masters thesis. University of California at Santa Cruz, December 1996.
- [4] M. Burrows and D. J. Wheeler. *A block-sorting lossless data compression algorithm*. 124. Digital Equipment Corporation Systems Research Center, Palo Alto, CA, 10 May 1994.
- [5] Michael Burrows, Charles Jerian, Butler Lampson, and Timothy Mann. On-line data compression in a log-structured file system. *Architectural Support for Programming Languages and Operating Systems* (Boston, MA, 12–15 October 1992). Published as *Computer Architecture News*, 20(special issue):2–9, October 1992.
- [6] Sailesh Chutani, Owen T. Anderson, Michael L. Kazar, Bruce W. Leverett, W. Anthony Mason, and Robert N. Sidebotham. The Episode File System. *Annual USENIX Technical Conference* (San Francisco, CA), pages 43–60, Winter 1992.
- [7] Dorothy Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, SE-13(2):222–232, February 1987.
- [8] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. *Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, 3–7 October 1998). Published as *SIGPLAN Notices*, 33(11):92–103, November 1998.
- [9] Garth A. Gibson, David F. Nagle, William Courtright II, Nat Lanza, Paul Mazaitis, Marc Unangst, and Jim Zelenka. NASD scalable storage systems. *USENIX.99* (Monterey, CA., June 1999), 1999.
- [10] Robert Hagmann. Reimplementing the Cedar file system using logging and group commit. *ACM Symposium on Operating System Principles* (Austin, Texas, 8–11 November 1987). Published as *Operating Systems Review*, 21(5):155–162, November 1987.
- [11] David Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. *Winter USENIX Technical Conference* (San Francisco, CA). Published as *Proceedings of USENIX*, pages 235–246. USENIX Association, 19 January 1994.
- [12] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.
- [13] James E. Johnson and William A. Laing. Overview of the Spirallog file system. *Digital Technical Journal*, 8(2):5–14, 1996.
- [14] Jeffrey Katcher. *PostMark: a new file system benchmark*. TR3022. Network Appliance, October 1997.
- [15] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A case for intelligent disks (IDISks). *SIGMOD Record*, 27(3):42–52, September 1998.
- [16] Gene H. Kim and Eugene H. Spafford. The design and implementation of Tripwire: a file system integrity

- checker. *Conference on Computer and Communications Security* (Fairfax, Virginia), pages 18–29, 2–4 November 1994.
- [17] Edward K. Lee and Chandramohan A. Thekkath. Petal: distributed virtual disks. *Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA). Published as *SIGPLAN Notices*, **31**(9):84–92, 1–5 October 1996.
 - [18] Christopher Lumb, Jiri Schindler, Gregory R. Ganger, David F. Nagle, and Erik Riedel. Towards higher disk head utilization: Extracting “free” bandwidth from busy disk drives. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000). ACM, October 2000.
 - [19] Josh MacDonald. *File system support for delta compression*. Masters thesis. Department of Electrical Engineering and Computer Science, University of California at Berkeley, 2000.
 - [20] Josh MacDonald, Paul N. Hilfinger, and Luigi Semenzato. PRCS: The project revision control system. *European Conference on Object-Oriented Programming* (Brussels, Belgium, July, 20–21). Published as *Proceedings of ECOOP*, pages 33–45. Springer-Verlag, 1998.
 - [21] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the performance of log-structured file systems with adaptive methods. *ACM Symposium on Operating System Principles* (Saint-Malo, France, 5–8 October 1997). Published as *Operating Systems Review*, **31**(5):238–252. ACM, 1997.
 - [22] K. McCoy. *VMS file system internals*. Digital Press, 1990.
 - [23] Marshall K. McKusick, William N. Joy, Samuel J. Lefler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, **2**(3):181–197, August 1984.
 - [24] *Object based storage devices: a command set proposal*. Technical report. October 1999. <http://www.T10.org/>.
 - [25] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from Bell Labs. *UKUUG Summer* (London), pages 1–9. United Kingdom UNIX systems User Group, Buntingford, Herts, 9–13 July 1990.
 - [26] Erik Riedel and Garth Gibson. *Active disks—remote execution for network-attached storage*. TR CMU-CS-97-198. December 1997.
 - [27] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, **10**(1):26–52, February 1992.
 - [28] Douglas J. Santry, Michael J. Feeley, and Norman C. Hutchinson. Elephant: the file system that never forgets. *Hot Topics in Operating Systems* (Rio Rico, AZ, 29–30 March 1992). IEEE Computer Society, 1999.
 - [29] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Ross W. Carton, Jacob Ofir, and Alistair C. Veitch. Deciding when to forget in the Elephant file system. *ACM Symposium on Operating System Principles* (Kiawah Island Resort, South Carolina). Published as *Operating Systems Review*, **33**(5):110–123. ACM, 12–15 December 1999.
 - [30] Margo I. Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. File system logging versus clustering: a performance comparison. *Annual USENIX Technical Conference* (New Orleans), pages 249–264. Usenix Association, 16–20 January 1995.
 - [31] Margo I. Seltzer, Gregory R. Ganger, M. Kirk McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems. *USENIX Annual Technical Conference* (San Diego, CA), 18–23 June 2000.
 - [32] M. Spasojevic and M. Satyanarayanan. An empirical study of a wide-area distributed file system. *ACM Transactions on Computer Systems*, **14**(2):200–222, May 1996.
 - [33] Adam Sweeney. Scalability in the XFS file system. *USENIX*. (San Diego, California), pages 1–14, 22–26 January 1996.
 - [34] Werner Vogels. File system usage in Windows NT 4.0. *ACM Symposium on Operating System Principles* (Kiawah Island Resort, Charleston, South Carolina, 12–15 December 1999). Published as *Operating System Review*, **33**(5):93–109. ACM, December 1999.
 - [35] Randolph Y. Wang, David A. Patterson, and Thomas E. Anderson. Virtual log based file systems for a programmable disk. *Symposium on Operating Systems Design and Implementation* (New Orleans, LA, 22–25 February 1999), pages 29–43. ACM, Winter 1998.
 - [36] Tatu Ylonen. SSH — Secure login connections over the internet. *USENIX Security Symposium* (San Jose, CA). USENIX Association, 22–25 July 1996.

Fast and secure distributed read-only file system

Kevin Fu, M. Frans Kaashoek, David Mazières

{fubob, kaashoek}@lcs.mit.edu, dm@cs.nyu.edu

MIT Laboratory for Computer Science

545 Technology Square

Cambridge, MA 02139

<http://www.fs.net/>

Abstract

Internet users increasingly rely on publicly available data for everything from software installation to investment decisions. Unfortunately, the vast majority of public content on the Internet comes with no integrity or authenticity guarantees. This paper presents the self-certifying read-only file system, a content distribution system providing secure, scalable access to public, read-only data.

The read-only file system makes the security of published content independent from that of the distribution infrastructure. In a secure area (perhaps off-line), a publisher creates a digitally-signed database out of a file system's contents. The publisher then replicates the database on untrusted content-distribution servers, allowing for high availability. The read-only file system protocol furthermore pushes the cryptographic cost of content verification entirely onto clients, allowing servers to scale to a large number of clients. Measurements of an implementation show that an individual server running on a 550 Mhz Pentium III with FreeBSD can support 1,012 connections per second and 300 concurrent clients compiling a large software package.

1 Introduction

This paper presents the design and implementation of a distributed file system that allows a large number of clients to access public, read-only data securely. Read-only data can have high performance,

availability, and security needs. Some examples include executable binaries, popular software distributions, bindings from hostnames to addresses or public keys, and popular, static Web pages. In many cases, people widely replicate and cache such data to improve performance and availability—for instance, volunteers often set up mirrors of popular operating system distributions. Unfortunately, replication generally comes at the cost of security. Each replica adds a new opportunity for attackers to break in and tamper with data, or even for the replica's own administrator to maliciously serve modified data.

People have introduced a number of ad hoc mechanisms for dealing with the security of public data, but these mechanisms often prove incomplete and of limited utility to other applications. For instance, binary distributions of Linux software packages in RPM [28] format can contain PGP signatures. However, few people actually check these signatures, and packages cannot be revoked. In addition, when packages depend on other packages being installed first, the dependencies cannot be made secure (e.g., one package cannot explicitly require another package to be signed by the same author). As another example, names of servers are typically bound to public keys through digitally signed certificates issued by a trusted authority. These certificates are distributed by the servers they authenticate, which naturally allows scaling to large numbers of servers. However, this approach also results in certificates having a long duration, which complicates revocation to the point that in practice many systems omit it.

To distribute public, read-only data securely, we have built a high-performance, secure, read-only file system designed to be widely replicated on untrusted servers. We chose to build a file system because of the ease with which one can refer to the file namespace in almost any context—from shell scripts to C

This research was partially supported by a National Science Foundation (NSF) Young Investigator Award, a USENIX scholars fellowship, and the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory under agreement number F30602-97-2-0288.

code to a Web browser's location field. Thus, the file system can support a wide range of applications, such as certificate authorities, that one could not ordinarily implement using a network file system.

Each read-only file system has a public key associated with it. We use the naming scheme of SFS [17], in which file names contain public keys. Thus, users can employ any of SFS's various key management techniques to obtain the public keys of file systems.

In our approach, an administrator creates a database of a file system's contents and digitally signs it off-line using the file system's private key. The administrator then widely replicates the database on untrusted machines. There, a simple and efficient server program serves the contents of the database to clients, without needing access to the file system's private key. DNS round-robin scheduling or more advanced techniques can be used to distribute the load between multiple replicas. A trusted program on the client machine checks the authenticity of data before returning it to the user.

The read-only file system avoids performing any cryptographic operations on servers and keeps the overhead of cryptography low on clients. We accomplish this with two simple techniques. First, blocks and inodes are named by *handles*, which are collision-resistant cryptographic hashes of their contents. Second, groups of handles are hashed recursively, producing a tree of hashes. Inodes contain the handles of a file's blocks. Directory blocks contain lists of file name to handle bindings. Using the handle of the root inode of a file system, a client can verify the contents of any block by recursively checking hashes.

The protocol between the client and server consists of only two remote procedure calls: one to fetch the signed handle for the root inode of a file system, and one to fetch the data (inode or file content) for a given handle. Since the server does not have to understand what it is serving, its implementation is both trivial and highly efficient: it simply looks up handles in the database and sends them back to the client.

We named the file system presented in this paper the *SFS read-only file system* because it uses SFS's naming scheme and fits into the SFS framework. The server-side of the file system consists of two programs: *sfsrodb* for creating signed databases off-line, and *sfsrosd* for serving signed databases from

untrusted machines. The client-side software consists of a daemon, *sfsrocd*, that queries databases through *sfsrosd* processes, verifies the results, and translates them into a file system.

A performance evaluation shows that *sfsrosd* can support 1012 short-lived connections per second on a PC (a 550 Mhz Pentium III with 256 Mbyte of memory) running FreeBSD, which is 26 times better than a standard read-write SFS file server and 92 times better than a secure web server. In fact, the performance of the read-only server is limited mostly by the number of TCP connections per second, not by the overhead of cryptography, which is offloaded to clients. For applications like sustained downloads that require longer-lived connections, *sfsrosd* can support 300 concurrent sessions while still saturating a fast Ethernet.

The rest of this paper is organized as follows. Section 2 relates our design to previous work. Section 3 details the design of the read-only server. Section 4 describes its implementation. Section 5 presents the applications of the read-only server. Section 6 evaluates the performance of these application and compares them to existing approaches. Section 7 concludes.

2 Related Work

We are unaware of a read-only (or read-write) file system that can support a high number of simultaneous clients and provide strong security. Many sites use a separate file system to replicate and export read-only binaries, providing high availability and high performance. AFS supports read-only volumes to achieve replication [22]. However, in all these cases replicas are stored on trusted servers. Some file systems provide high security (e.g., the SFS read-write file system [17] or Echo [2]), but compared to the SFS read-only file system these servers do not scale well with the number of clients because their servers perform expensive cryptographic operations in the critical path (e.g., the SFS read-write server performs one private-key operation per client connection, which takes about 24 msec on a 550 Mhz Pentium III).

Secure DNS [8] is an example of a read-only data service that provides security, high availability, and high performance. In secure DNS, each individual resource record is signed. This approach does not

work for file systems. If each inode and 8 Kbyte-block of a moderate file system—for instance, the 635 Mbyte Red Hat 6.2 i386 distribution—had to be signed individually with a 1,024-bit key, the signing alone would take about 36 minutes ($90,000 \times 24$ msec) on a 550 Mhz Pentium III. A number of read-only data services, such as FTP archives, are 100 times bigger, making individual block signing impractical—particularly since we want to allow frequent updates of the database and rapid expiration of old signatures.

Secure HTTP servers are another example of servers that provide access to mostly read-only data. These servers are difficult to replicate on untrusted machines, however, since their private keys have to be on-line to prove their identity to clients. Furthermore, private-key operations are expensive and are in the critical path: every SSL connection requires the server to compute modular exponentiations as part of the public-key cryptography [11]. As a result, software-only secure Web servers achieve low throughput (with a 1,024-bit key, IE and Netscape servers can typically support around 15 connections per second).

Content distribution networks built by companies such as Adero, Akamai, Cisco, and Digital Island are an efficient and highly-available way of distributing static Web content. Content stored on these networks is dynamically replicated on trusted caches scattered around the Internet. Web browsers then connect to a cache that provides high performance. The approach described in this paper would allow read-only Web content to be replicated securely to *untrusted* machines and would provide strong data integrity to clients that run our software. For clients that don't run our software, one can easily configure any Web server on an SFS client to serve the `/sfs` directory, trivially creating a Web-to-SFS gateway for any Web clients that trust the server.

Signed software distributions are common in the open-source community. In the Linux community, for example, a creator or distributor of a software package can sign RPM [28] files with PGP or GNU GPG. RPM also supports MD5 hashes. A person downloading the software can optionally check the signature. Red Hat Software, for example, publishes their PGP public key on their Web site and signs all their software distributions with the corresponding private key. This setup provides some guarantees to the person who checks the signature on the RPM file and who makes sure that the public key indeed belongs to Red Hat. However, RPMs do not provide

an expiration time or revocation support. If users were running the SFS client software and RPMs were stored on SFS read-only file servers, the server would be authenticated transparently and the data would be checked transparently for integrity and recentness.

The read-only file system makes extensive use of hash trees, which have appeared in numerous other systems. Merkle used a hierarchy of hashes for an efficient digital signature scheme [18]. In the context of file systems, the Byzantine-fault-tolerant file system uses hierarchical hashes for efficient state transfers between clients and replicas [5, 6]. The cryptographic storage file system [12] uses cryptographic hashes in a similar fashion to the SFS read-only file system. Duchamp uses hierarchical hashes to efficiently compare two file systems in a toolkit for partially-connected operation [7]. TDB [16] uses hash trees combined with a small amount of trusted storages to construct a trusted database system on untrusted storage. Finally, a version of a network-attached storage device uses an incremental “Hash and MAC” scheme to reduce the cost of protecting the integrity of read traffic in storage devices that are unable to generate a MAC at full data transfer rates [14].

A number of proposals have been developed to make digital signatures cheaper to compute [13, 20], some involving hash trees [27]. These proposals enable signing hundreds of packets per second in applications such as multicast streams. However, if applied to file systems, these techniques would introduce complications such as increased signature size. Moreover, because the SFS read-only file system was designed to avoid trusting servers, read-only servers must function without access to a file system's private key. This prevents any use of dynamically computed digital signatures, regardless of the computational cost.

3 SFS read-only file system

Figure 1 shows the overall architecture of the SFS read-only file system. In a secure area, an administrator runs the SFS read-only database generator (`sfsrodb`), passing as arguments a directory of files to export and a file containing a private key. The administrator replicates this database on a number of untrusted machines, each of which runs a copy of the SFS read-only server daemon (`sfsrosd`). The

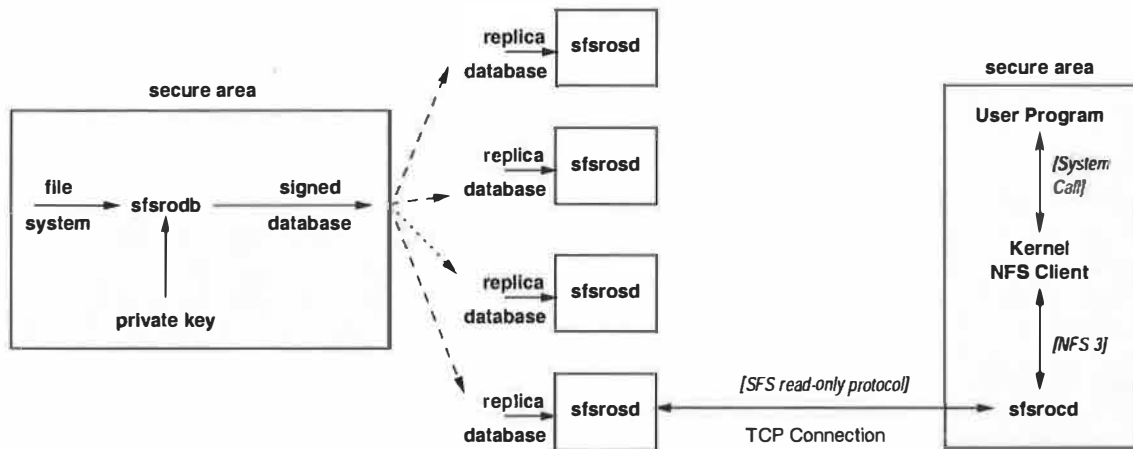


Figure 1: The SFS read-only file system. Shaded boxes show the trusted computing base.

read-only server is a simple program that looks up the data for a given handle in the replica's database and returns the data to the client.

Most of the actual file system is implemented by the SFS read-only client daemon (`sfsrocd`), which runs on the client's machine. It handles file system requests from the local operating system and responds to them. The read-only client understands the format of inodes and directories. It parses pathnames, searches directories, looks up blocks of files, etc.

In order to respond to requests from the local operating system, the client retrieves data from one of the servers that has a replica of the database. DNS round-robin scheduling or more advanced techniques (e.g., [15]) can be used to select a replica that provides good performance. Since the replica may run on untrusted hardware, the client must verify that any data sent by the server was indeed signed by a database generator with the appropriate private key.

The SFS read-only file system assumes that an attacker may compromise and assume control of any read-only server machine. It therefore cannot prevent denial-of-service from an attacker penetrating and shutting down every server for a given file system. However the client does ensure that any data retrieved from a server is authentic, no older than a file system-configurable consistency period, and also no older than any previously retrieved data from the same file system. The read-only file system does not provide confidentiality. Thus, data on replicas does not have to be kept secret from attackers. The key security property of the read-only file system is in-

Operation	Cost (μ sec)
Sign 68 byte fsinfo	24,400
Verify 68 byte fsinfo	82
SHA-1 256 byte iv+inode	17
SHA-1 8,208 byte iv+block	406

Figure 2: Performance of base primitives on a 550 Mhz Pentium III. Signing and verification use 1,024-bit Rabin-Williams keys.

tegrity.

Our design can also in principle be used to provide non-repudiation of file system contents. An administrator of a server could commit to keeping every file system he ever signed. Then, clients could just record the signed root handle. The server would be required to prove what the file system contained on any previous day. In this way, an administrator could never falsely deny that a file previously existed.

Figure 2 lists the cryptographic primitives that we use in the read-only file system. We chose the Rabin public key cryptosystem [26] for its fast signature verification time. The implementation is secure against chosen-message attacks (using the redundancy function proposed in [1]). As can be seen from Table 2, computing digital signatures is somewhat expensive, but verifying them is takes only 82 μ sec—far cheaper than a typical network round trip time, in fact.

SFS also uses the SHA-1 [9] cryptographic hash function. SHA-1 is a collision-resistant hash function that produces a 20-byte output from an arbitrary-length input. Finding any two inputs of SHA-1 that


```

struct FSINFO {
    sfs_time start;
    unsigned duration;
    opaque iv[16];
    sfs_hash rootfh;
    sfs_hash fhdb;
};

```

Figure 3: Contents of the digitally signed root of an SFS read-only file system.

produce the same output is believed to be computationally intractable. Modern machines can typically compute SHA-1 at a rate greater than the local area network bandwidth. Thus, one can reasonably hash the result of every RPC in a network file system protocol.

The rest of this section describes how we use these primitives to efficiently provide authenticity and recency of data.

3.1 SFS read-only Protocol

The read-only protocol uses two RPCs: *getfsinfo* and *getdata*. *getfsinfo* takes no arguments and returns a digitally signed FSINFO structure, depicted in Figure 3. The SFS client verifies the signature using the public key embedded in the server's name. The *getdata* RPC takes a 20-byte hash value as an argument and returns a data block producing that hash value. The client uses *getdata* to retrieve parts of the file system requested by the user, and verifies the authenticity of the blocks using the FSINFO structure.

Because read-only file systems may reside on untrusted servers, the protocol relies on time to enforce consistency loosely but securely. The *start* field of FSINFO indicates the time (in seconds since 1970) at which a file system was signed. Clients cache the highest value they have seen to prevent an attacker from rolling back the file system to a previous version. The *duration* field signifies the length of time for which the data structure should be considered valid. It represents a commitment on the part of a file system's owner to issue a newly signed file system within a certain period of time. Clients reject an FSINFO structure when the current time exceeds *start* + *duration*.

The file system names arbitrary-length blocks of data with fixed-size handles. The handle for a

data item x is computed using SHA-1: $H(x) = \text{SHA-1}(iv, x)$. *iv*, the initialization vector, is randomly chosen by *sfsrodb* the first time the administrator creates a database for a file system. It ensures that simply knowing one particular collision of SHA-1 will not immediately give attackers collisions of functions actually used by SFS file systems.

rootfh is the handle of the file system's root directory. It is a hash of the root directory's *inode* structure, which through recursive use of H specifies the contents of the entire file system, as described below. *fhdb* is the the hash of the root of a tree that contains every handle reachable from the root directory. *fhdb* lets clients securely verify that a particular handle does not exist, so that they can return stale file handle errors when file systems change. *fhdb* will not be necessary in future versions of the software, as described in Section 3.4.

3.2 SFS read-only inode

Figure 4 shows the format of an inode in the read-only file system. The inode begins with some metadata, including the file's type (regular file, executable file, directory, opaque directory, or symbolic link), size, and modification time. Permissions are not included because they can be synthesized on the client. The inode then contains handles of successive 8 Kbyte blocks of file data. If the file contains more than eight blocks, the inode contains the handle of an *indirect block*, which in turn contains handles of file blocks. Similarly, for larger files, an inode can also contain the handles of double- and triple-indirect blocks. In this way, the blocks of small files can be verified directly from the inode, while inodes can also indirectly verify large files—an approach similar to the on-disk data structures of the Unix File System [19].

3.3 Database generator

To export a file system, a system administrator produces a signed database from a source directory in an existing file system. The database contains file data blocks and inodes indexed by their hash values. In essence, it is analogous to a file system in which inode and block numbers have been replaced by cryptographic hashes.

The database generator utility traverses the given file system depth-first to builds the database. The

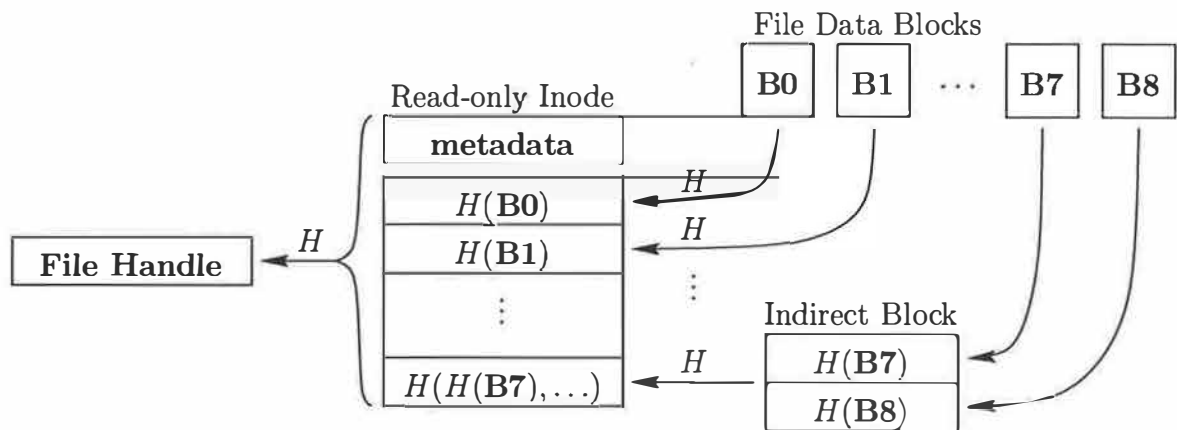


Figure 4: Format of a read-only file system inode.

leaves of the file system tree are files or symbolic links. For each regular file in a directory, the database generator creates a read-only inode structure and fills in the metadata. Then, it reads the blocks of the file. For each block, `sfsrodb` hashes the data in that block to compute its handle, and then inserts the block into the database under the handle (i.e., a lookup on the handle will return the block). The hash value is also stored in an inode. When all file blocks of a file are inserted into the database, the filled-out inode is inserted into the database under its hash value.

When all files in a given directory have been inserted into the database, the generator utility inserts a file corresponding to the directory itself. The file blocks of a directory contain lists of (name, handle) pairs; the directory's inode contains hashes of those blocks (and possibly indirect blocks) as for regular files. Directories are sorted lexicographically by name. Thus, clients can avoid traversing the entire directory by performing a binary search when looking up files in very large directories (e.g., a directory that contains all names in the `.com` domain). This property also allows clients to verify inexpensively whether a file name exists or not, without having to read the whole directory.

Each directory also contains its full pathname from the root of the file system. The client uses the pathname to evaluate the file name `".."` locally, using it as a reference for any directory's parent. (Since a directory inode's handle depends on the handles of all subdirectories, a circular dependency makes it impossible to create directory entries of the form `("..", parent's handle)`.) Clients verify that a directory contains the proper pathname when first

looking it up. This is not strictly necessary—an administrator signing a bad database should expect undefined interpretations by clients. However, the sanity check reduces potentially confusing behavior on clients of malformed file systems.

Inodes for symbolic links are slightly different from the one depicted in Figure 4. Instead of containing handles of blocks, the inode directly contains the destination path for the symbolic link.

To avoid inconveniencing users with large directories, server administrators can set the type field in an inode to "opaque directory." When users list an opaque directory, they see only entries they have already referenced—somewhat like Unix "automounter" directories [4]. Opaque directories are well-suited to giant directories containing, for instance, all names in the `.com` domain or all all name-to-key bindings issued by a particular certificate authority. If one used non-opaque directories for these applications, users could inadvertently download hundreds of megabytes of directory data by typing `ls` or using file name completion in the wrong directory.

After the whole directory tree has been inserted into the database, the generator utility fills out an `FSINFO` structure and signs it with the private key of the file system. For simplicity, `sfsrodb` stores the signed `FSINFO` structure in the database under a well-known, reserved key.

The database generator stores inodes, directories, and file block data in the database in XDR marshaled form [24]. Using XDR has three advantages. First, it simplifies the client implementation, as the

client can use the SFS RPC and crypto libraries to parse file system data. Second, the XDR representation clearly defines what the database contains, which simplifies writing programs that process the database (e.g., a debugging program). Finally, it improves performance of the read-only server by saving it from having to do any marshaling.

3.4 Updating file systems

The biggest challenge in updating read-only file systems is dealing with data that no longer exists in the file system. When a file system changes, the administrator generates a new database and pushes it out to the server replicas. Files that persist across file system versions will keep the same handles. However, when a file is removed or modified, clients can end up requesting handles no longer in the database. In this case, the read-only server replies with an error.

Unfortunately, since read-only servers (and the network) are not trusted, clients cannot necessarily believe “handle not found” errors they receive. Though a compromised server can hang a client by refusing to answer RPCs, it must not be able to make programs spuriously abort with stale file handle errors. Otherwise, for instance, an application looking up a key revocation certificate in a read-only file system might falsely believe that the certificate did not exist.

We have two schemes to let clients securely determine whether a given file handle exists: the current scheme uses the `fhdb` field of the `FSINFO` structure to verify that a handle no longer exists. `fhdb` is the root of a hash tree, the leaf nodes of which contain a sorted list of every handle in the file system. Thus, clients can easily walk the hash tree (using *getfsinfo*) to see whether the database contains a given file handle.

The `fhdb` scheme has advantages. It allows files to persist in the database even after they have been deleted, as not every handle in the database need be reachable from the root directory. Thus, by keeping handles of deleted files in a few subsequent revisions of a database, a system administrator can support the traditional Unix semantics that one can continue to access an open file even after it has been deleted.

Unfortunately, `fhdb` has several drawbacks. Even small changes to the file system cause most of the

hash tree under `fhdb` to change (making incremental database updates unnecessarily expensive). Furthermore, in the read-only file system, because handles are based on file contents, there is no distinction between modifying a file and deleting then recreating it. In some situations, one doesn’t want to have to close and reopen a file to see changes. (This is always the case for directories, which therefore need a different mechanism anyway.) Finally, under the `fhdb` scheme, a server cannot change its `iv` without causing all open files to become stale on all clients.

To avoid these problems, future versions of the software will eliminate `fhdb`. Instead, the client will track the pathnames of all files accessed in read-only file systems. When a server `FSINFO` structure is updated, the client will walk the file namespace to find the new inode corresponding to the name of each open file. Those who really want an open file never to change can still emulate the old semantics (albeit somewhat inelegantly) using a symbolic link to switch between the old and new version of a file while allowing both to exist simultaneously. Once clients track the pathnames of files, directories need no longer contain their full pathnames: clients will have enough state to evaluate the parent directory name “..” on their own.

The read-only inode structure contains the modification and “inode change” times of a file. Thus, `sfsrodb` could potentially update the database incrementally after changes are made to the file system, recomputing only the hashes from changed files up to the root handle and the signature on the `FSINFO` structure. Our current implementation of `sfsrodb` creates a completely new database for each version of the file system, but we plan to support incremental updates in a future release.

3.5 Incremental transfer

We built a simple utility program, `pulldb`, that incrementally transfers a newer version of a database from one replica to another. The program fetches `FSINFO` from the source replica, and checks if the local copy of the database is out of date. If so, the program recursively traverses the entire file system, starting from the new root file handle, building on the side a list of all active handles. For each handle encountered, if the handle does not already exist in the local database, `pulldb` fetches the corresponding data with a *getdata* RPC and stores it in database. After the traversal, `pulldb` swaps the `FSINFO` structure in the database and then deletes all handles no

longer in the file system. If a failure appears before the transfer is completed, the program can just be restarted, since the whole operation is idempotent.

3.6 Read-only server

The server program `sfsrosd` is a trivial—only 400 lines of C++. `sfsrosd` knows nothing about the structure of the file system it serves; it simply gets requests for handles, looks up the handles in the database, and returns their values to the client. It also fields `getfsinfo` and `SFS connect` RPCs, to which it replies with two static data structures cached in memory.

3.7 Read-only client

The client program constitutes the bulk of the code in the read-only file system (1,500 lines of C++). The read-only client behaves like an NFS3 [3] server, allowing it to communicate with the operating system through ordinary networking system calls. The read-only client resolves pathnames for file name lookups and handles reads of files, directories, and symbolic links. It relies on the server only for serving blocks of data, not for interpreting or verifying those blocks. The client checks the validity of all blocks it receives against the hashes by which it requested them.

We demonstrate how the client works by example. Consider a user reading the file `/sfs/sfs.mit.edu:bzcc5hder7cuc86kf6qswyx6yuemn w69/README`, where `bzcc5hder7cuc86kf6qswyx6yuemn w69` is the representation of the public key of the server storing the file `README`. (In practice, symbolic links save users from ever having to see or type pathnames like this.)

The local operating system's NFS client will call into the protocol-independent SFS client software, asking for the directory `/sfs/sfs.mit.edu:bzcc5hder7cuc86kf6qswyx6yuemn w69/`. The client will contact `sfs.mit.edu`, which will respond that it implements the read-only file system protocol. At that point, the protocol-independent SFS client daemon will pass the connection off to the read-only client, which will subsequently be asked by the kernel to interpret the file named `README`.

The client makes a `getfsinfo` RPC to the server to get the file system's signed `FSINFO` structure. It

verifies the signature on the structure, ensures that the `start` field is no older than its previous value if the client has seen this file system before, and ensures that `start + duration` is in the future.

The client then obtains the root directory's inode by doing a `getdata` RPC on the `rootfh` field of `FSINFO`. Given that inode, it looks up the file `README` by doing a binary search among the blocks of the directory, which it retrieves through `getdata` calls on the block handles in the directory's inode (and possibly indirect blocks). When the client has the directory entry `(README, handle)`, it calls `getdata` on handle to obtain `README`'s inode. Finally, the client can retrieve the contents of `README` by calling `getdata` on the block handles in its inode.

4 Implementation

As illustrated in Figure 5, the read-only file system is implemented as two new daemons (`sfsrocd` and `sfsrosd`) in the SFS system [17]. `sfsrodb` is a stand-alone program.

`sfsrocd` and `sfsrosd` communicate with Sun RPC over a TCP connection. (The exact message formats are described in the XDR protocol description language [24].) We also use XDR to define cryptographic operations. Any data that the read-only file system hashes or signs is defined as an XDR data structure; SFS computes the hash or signature on the raw, marshaled bytes.

`sfsrocd`, `sfsrosd`, and `sfsrodb` are written in C++. To handle many connections simultaneously, the client and server use SFS's asynchronous RPC library. Both programs are single-threaded, but the RPC library allows the client to have many outstanding RPCs.

Because of SFS's support for developing new servers and `sfsrosd`'s simplicity, the implementation of `sfsrosd` is trivial. It gets requests for data blocks by file handle, looks up preformatted responses in a B-tree, and responds to the client. The current implementation uses the Sleepycat database's B-tree [23]. In the measurements `sfsrosd` accesses the database synchronously.

The implementations of the other two programs (`sfsrodb` and `sfsrocd`) are more interesting; we discuss them in more detail.

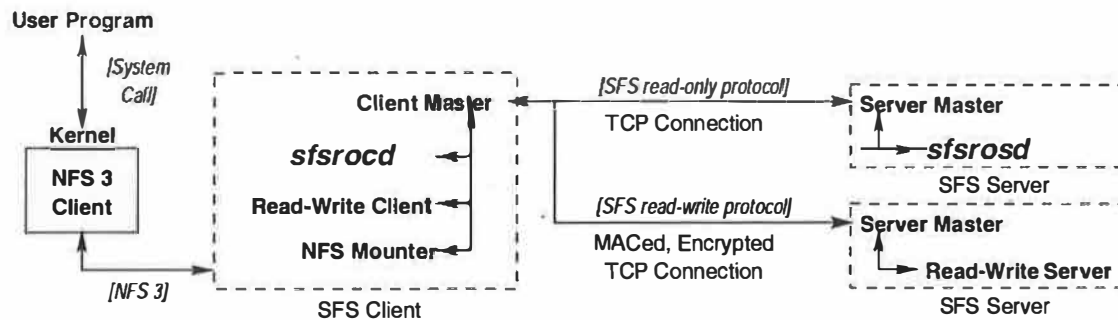


Figure 5: Implementation overview of the read-only file system in the SFS framework.

4.1 sfsrodb

The implementation of **sfsrodb** is simple. It is a short stand-alone C++ program. It walks down the given file system and builds a database with file system data indexed by the cryptographic hash of the data. After building the database, it creates an **FSINFO** structure and signs it with the private key of the file system.

The disadvantage of storing the data in marshaled form is that physical representation of the data is slightly larger than the actual data. For instance, an 8 Kbyte file block is slightly larger than 8 Kbyte. The Sleepycat database does not support values just over a power of 2 in size very well; we are developing a light-weight, asynchronous B-tree that handles such odd-sized values well.

A benefit of storing blocks under their hash is that blocks from different files that have the same hash will only be stored once in the database. If a file system contains blocks with identical content among multiple files, then **sfsrodb** stores just one block under the hash. In the RedHat 6.2 distribution, 5,253 out of 80,508 file data blocks share their hash with another block. The overlap is much greater if one makes the same data available in two different formats (for instance, the contents of the RedHat 6.2 distribution, and the image of a CD-ROM containing that distribution).

4.2 sfsrocd

sfsrocd implements four caches with LRU replacement policies to improve performance by avoiding RPCs to **sfsrocd**. It maintains an inode cache, an indirect-block cache, a small file-block cache, and a cache for directory entries.

sfsrocd's small file-block cache primarily optimizes the case of the same block appearing in multiple files. In general, **sfsrocd** relies on the local operating system's buffer cache to cache the file contents. Thus, any additional caching of file contents will tend to waste memory unless a block appears in multiple places. The small block cache optimizes common cases—such as a file with many blocks of all zeros—without dedicating too much memory to redundant caching.

Indirect blocks are cached so that **sfsrocd** can quickly fetch and verify multiple blocks from a large file without refetching the indirect blocks. **sfsrocd** does not prefetch because most operating systems already implement prefetching locally.

5 Applications

To demonstrate the usefulness of the SFS read-only file system, we describe two applications that we measure in Section 6: certificate authorities and software distribution.

5.1 Certificate Authorities

Certificate authorities for the Internet are servers that publish certificates binding hostnames to public keys. On the Web, for instance, the certificate authority Verisign certifies server keys for Web servers. Verisign signs the domain name and the public key of the Web server in an X.509 certificate, and returns this to the Web server administrator [10]. When a browser connects to the Web server with secure HTTP, the server responds with the certificate. The browser checks the validity of the certificate by verifying it with Verisign's public key. Most popular

browsers have Verisign's key embedded in their binaries. One benefit of this approach is that Verisign does not have to be on-line when the browser connects to a certified Web server. However, this comes at the cost of complicating certificate revocation to the point that in practice no one does it.

In contrast, SFS uses file systems to certify public keys of servers. SFS certificate authorities are nothing more than ordinary file systems serving symbolic links that translate human-readable names into public keys that name file servers [17]. For example, if Verisign acted as an SFS certificate authority, client administrators would likely create symbolic links from their local disks, for instance `/verisign`, to Verisign's self-certifying pathname—a pathname containing the public key of Verisign's file system. This file system would in turn contain symbolic links to other SFS file systems. For example, `/verisign/NYU` might be a symbolic link to a self-certifying pathname for an SFS file server that Verisign calls NYU.

Unlike traditional certificate authorities, SFS certificate authorities get queried interactively. This simplifies certificate revocation, since revoking a key amounts to removing the symbolic link. However, it also places high integrity, availability, and performance demands on file systems serving as on-line certificate authorities.

By running certificate authorities as SFS read-only file systems, we can address these needs. The SFS read-only file system improves performance by making the amount of cryptographic computation proportional to the file system's size and rate of change, rather than to the number of clients connecting. SFS read-only also improves integrity by freeing SFS certificate authorities from the need to keep any on-line copies of their private keys. Finally, SFS read-only improves availability because it can be replicated on untrusted machines.

An administrator adds certificates to its SFS file system by adding new symbolic links. The database is updated once a day, similarly to second-level DNS updates. The administrator (incrementally) replicates the database to other servers.

The certificate authority database (and thus its certificates) might be valid for one day. The certificate that we bought from Verisign for our Web server is valid for 12 months. If the private key of an SFS server is compromised, then the next day the certificate will be out of the on-line database.

SFS certificate authorities also support key revocation certificates to revoke public keys of servers explicitly. The key revocation certificates are self-authenticating [17] and signed with the private key of the compromised server. Verisign could, for example, maintain an SFS certificate authority that has a directory to which users upload revocation certificates for some fee; since the certificates are self-authenticating, Verisign does not have to certify them. Clients check this directory when they perform an on-line check for key certificates. Because checks can be performed interactively, this approach works better than X.509 certificate revocation lists [10].

5.2 Software Distribution

Sites distributing popular software have high availability, integrity, and performance needs. Open software is often replicated at several mirrors to support a high number of concurrent downloads. If users download a distribution with anonymous FTP, they have low data integrity: a user cannot tell whether he is downloading a trojan-horse version instead of the correct one. If users connect through the Secure Shell (SSH) or secure HTTP, then the server's throughput is low because of cryptographic operations it must perform. Furthermore, that solution doesn't protect against attacks where the server is compromised and the attacker replaces a program on the server's disk with a trojan horse.

By distributing software through SFS read-only servers, one can provide integrity, performance, and high availability. Users with `sfsrocd` can even browse the distribution as a regular file system and compile the software straight from the sources stored on the SFS file system. `sfsrocd` will transparently check the authenticity of the file system data. To distribute new versions of the software, the administrator simply updates the database. Users with only a browser could get all the benefits by just connecting through a Web-to-SFS proxy to the SFS file system.

Software distribution using the read-only file systems complements signed RPMs. First, RPMs do not provide any revocation support; the signature on an RPM is good forever. Second, there is no easy way to determine whether an RPM is recent; an attacker can give a user an older version of a software package without the user knowing it. Third, there is no easy method for signing a collection of RPMs

that constitute a single system. For an example, there is currently no way of cryptographically verifying that one has the complete Linux RedHat 6.2 distribution (or all necessary security patches for a release). Using SFS read-only, Red Hat could securely distribute the complete 6.2 release, providing essentially the same security guarantees as a physical CDROM distribution.

6 Performance

This section presents the results of measurements to support the claims that (1) SFS read-only provides acceptable application performance and (2) SFS read-only scales well with the number of clients.

To support the first claim, we measure the performance of microbenchmarks and a large software compilation. We compare the performance of the SFS read-only file system with the performance on the local file system, insecure NFS, and the secure SFS read-write file system.

To support the second claim, we measure the maximum number of connections per server and the throughput of software downloads with an increasing number of clients.

We expect that the main factors affecting SFS read-only performance are the user-level implementation in the client, hash verification in the client, and database lookups on the server.

6.1 Experimental setup

We measured performance on 550 MHz Pentium IIIs running FreeBSD 3.3. The client and server were connected by 100 Mbit, full-duplex, switched Ethernet. Each machine had a 100 Mbit Tulip Ethernet card, 256 Mbytes of memory, and an IBM 18ES 9 Gigabyte SCSI disk. In `sfsrocd`, the inode, indirect-block, and directory entry caches each have a maximum of 512 entries, while the file-block cache has maximum of 64 entries. Maximum TCP throughput between client and server, as measured by `ttcp` [25], was 11.31 Mbyte/sec.

Because the certificate authority benchmark in Section 6.4 requires many CPU cycles on the client, we also employed two 700 MHz Athlons running OpenBSD 2.7. Each Athlon had a 100 Mbit Tulip

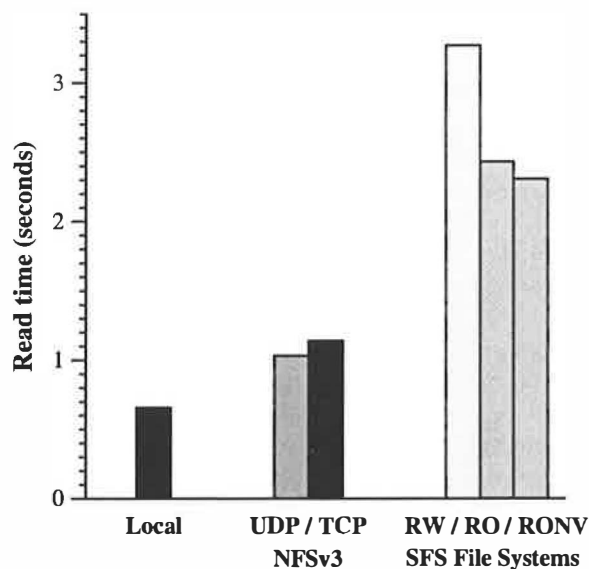


Figure 6: Time to sequentially read 1,000 1 Kbyte files. Local is FreeBSD's local FFS file system on the server. The local file system was tested with a cold cache. The network tests were applied to warmed server caches, but cold client caches. RW, RO, and RONV denote respectively the read-write protocol, the read-only protocol, and the read-only protocol with no verification.

Ethernet card and 128 Mbytes of memory. Maximum TCP throughput between an Athlon and the FreeBSD server, as measured by `ttcp`, was 11.04 Mbyte/sec. The Athlon machines generated the client SSL and SFSRW requests; we report the sum of the performance measured on the two machines.

For all experiments we report the average of five runs.

6.2 Microbenchmarks

To evaluate the performance of the SFS read-only system, we perform small and large file microbenchmarks.

6.2.1 Small file benchmark

We use the read phases of the LFS benchmarks [21] to obtain a basic understanding of single client/single server performance. Figure 6 shows the latency of sequentially reading 1,000 1 Kbyte

Breakdown	Cost (sec)
NFS loopback	0.661
Computation in client	1.386
Communication with server	0.507
Total	2.55

Table 1: Breakdown of SFS read-only performance reported in Fig 6.

files on the different file systems. The files contain random data and are distributed evenly across ten directories. For the read-only and NFS experiments, all samples were within 0.4% of the average. For the read-write experiment, all samples were within 2.7% of the average. For the local file system, all samples were within 6.9% (0.4 seconds) of the average.

As expected, the SFS read-only server performs better than the SFS read-write server (2.43 vs. 3.27 seconds). The read-only file server performs worse than NFSv3 over TCP (2.43 vs. 1.14 seconds). To understand the performance of the read-only file server, we break down the 2.43 seconds spent in the read-only client (see Table 1).

To measure the cost of the user-level implementation we measured the time spent in NFS loopback. We used the `fchown` operation against a file in a read-only file system to measure the time spent in the user-level NFS loopback file system. This operation generates NFS RPCs from the kernel to the read-only client, but no traffic between the client and the server. The average over 1000 `fchown` operations is 167 μ sec. By contrast, the average for attempting an `fchown` of a local file with permission denied is 2.4 μ sec. The small file benchmark generates 4015 NFS loopback RPCs. Hence, the overhead of the client's user-level implementation is at least $(167 \mu\text{sec} - 2.4 \mu\text{sec}) * 4015 = 0.661$ seconds.

We also measured the CPU time spent during the small file benchmark in the read-only client at 1.386 seconds. With verification disabled, this drops to 1.300 seconds, indicating that for this workload, file handle verification consumes very little CPU time.

To measure the time spent communicating with the read-only server, we timed the playback of a trace of the 2101 `getdata` RPCs of the benchmark to the read-only server. This took 0.507 seconds.

These three measurements total to 2.55 seconds. With an error margin of 5%, this accounts for the 2.43 seconds to run the benchmark. We attribute

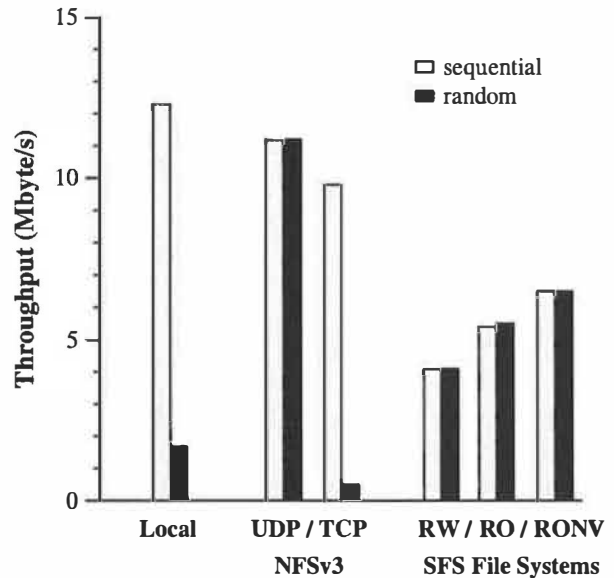


Figure 7: Throughput of sequential and random reads of a 40 Mbyte file. The experimental conditions are the same as in Figure 6.

this error to a small amount of double counting of cycles between the NFS loopback measurement and the computation in the client.

The cryptography accounts for very little of the time. The CPU time spent on verification is only 0.086 seconds. Moreover, end-to-end measurements show that data verification has little impact on performance. RONV performs slightly better than RO (2.31 vs. 2.43 seconds). Therefore, any optimization will have to focus on the non-cryptographic portions of the system.

6.2.2 Large file benchmark

Figure 7 shows the performance of sequentially and randomly reading a large (40 Mbyte) file containing random data. We read in blocks of 8 KBytes. In the network experiments, the file is in the server's cache, but not in the client's cache. Thus, we are not measuring the server's disk. This isolates the software overhead of cryptography and SFS's user-level design. For the local file system, all samples were within 1.4% of the average. For NFSv3 over UDP and the read-write experiments, all samples were within 1% of the average. For NFSv3 over TCP and the read-only experiments, all samples were within 4.3% of the average. This variability and the poor

NFSv3 over TCP performance appears to be due to a pathology of FreeBSD.

The SFS read-only server performs better than the read-write server because the read-only server performs no on-line cryptographic operations. On the sequential workload, verification costs 1.4 Mbyte/s in throughput. NFSv3 over TCP performs substantially better (9.8 vs. 6.5 Mbyte/s) than the read-only file system without verification, even though both run over TCP and do similar amounts of work; the main difference is that NFS is implemented in the kernel.

If the large file contains only blocks of zeros, SFS read-only obtains a throughput of 17 Mbyte/s since all blocks hash to the same handle. In this case, the measurement is dominated by the throughput of loop-back NFSv3 over UDP on the client machine.

6.3 Software distribution

To evaluate how well the read-only file system performs on a larger application benchmark, we compiled (with optimization and debugging disabled) Emacs 20.6 with a local build directory and a remote source directory. The results are shown in Figure 8. The RO experiment performs 1% worse (1 second) than NFSv3 over UDP and 4% better (3 seconds) than NFSv3 over TCP. Disabling integrity checks in the read-only file system (RONV) does not speed up the compile because our caches absorb the cost of hash verification. However, disabling caching does decrease performance (RONC). During a single Emacs compilation, the read-only server consumes less than 1% of its CPU while the read-only client consumes less than 2% of its CPU. This demonstrates that the read-only protocol introduces negligible performance degradation in an application benchmark.

To evaluate how well `sfsrostd` scales, we took a trace of a single client compiling the Emacs 20.6 source tree, repeatedly played the trace to the server from an increasing number of simulated, concurrent clients, and plotted the aggregate throughput delivered by `sfsrostd`. The results are shown in Figure 9. Each sample represents the throughput of playing traces for 100 seconds. Each trace consists of 1428 RPCs. With 300 simultaneous clients, the server consumes 96% of the CPU.

With more than 300 clients, the FreeBSD server reboots because of a bug in its TCP implementation.

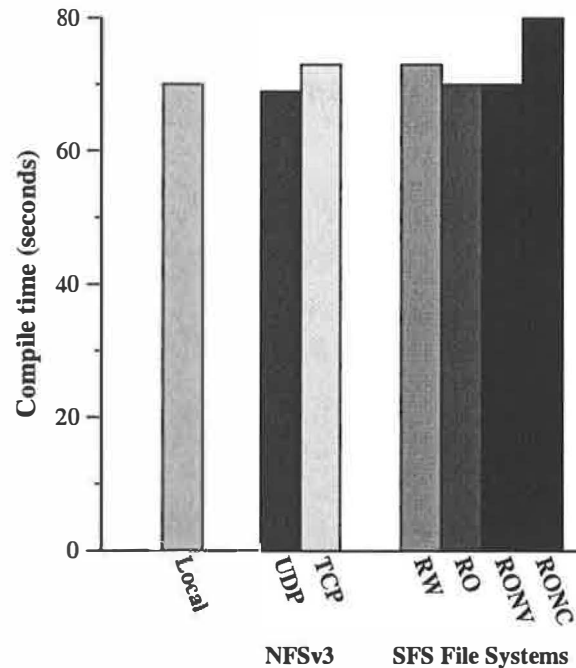


Figure 8: Compiling the Emacs 20.6 source. Local is FreeBSD's local FFS file system on the server. The local file system was tested with a cold cache. The network tests were applied to warmed server caches, but cold client caches. RW, RO, RONV, and RONC denote respectively the read-write protocol, the read-only protocol, the read-only protocol with no verification, and the read-only protocol with no caching.

We replaced the FreeBSD server with an OpenBSD server and measured that `sfsrostd` maintains a rate of 10 Mbyte/s of file system data up to 600 simultaneous clients.

6.4 Certificate authority

To evaluate whether the read-only file system performs well enough to function as an on-line certificate authority, we compare the number of connections a single read-only file server can sustain with the number of connections to the SFS read-write server, the number of SSL connections to an Apache web server, and the number of HTTP connections to an Apache server.

The SFS servers use 1024-bit keys. The SFS read-write server performs one Rabin-Williams decryption per connection while the SFS read-only server performs no on-line cryptographic operations. The

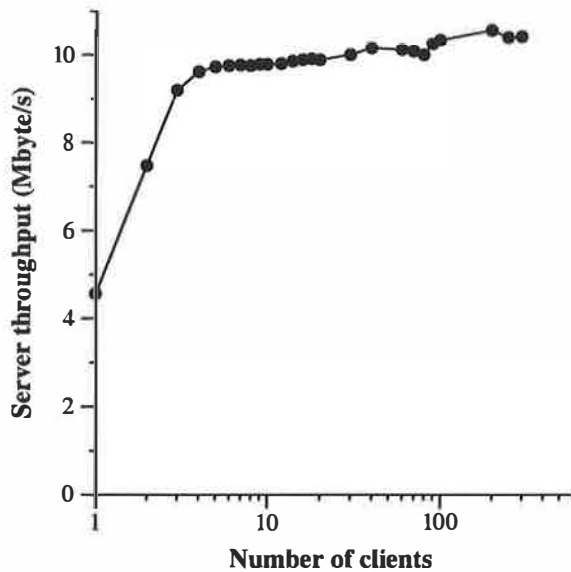


Figure 9: The aggregate throughput delivered by the read-only server for an increasing number of clients simultaneously compiling the Emacs 20.6 source. The number of clients is plotted on a log scale.

Web server was Apache 1.3.12 with OpenSSL 0.9.5a and ModSSL 2.6.3-1.3.12. Our SSL ServerID certificate and Verisign CA certificate use 1024-bit RSA keys. All the SSL connections use the TLSv1 cipher suite consisting of Ephemeral Diffie-Hellman key exchange, DES-CBC3 for confidentiality, and SHA-1 HMAC for integrity.

To generate enough load to saturate the servers, we wrote a simple client program that sets up connections, reads a small file containing a self-certifying path, and terminates the connection as fast as it can. We run this client program simultaneously on two OpenBSD machines. In all experiments, the certificate is in the main memory of the server, so we are limited by software performance, not by disk performance. This scenario is realistic since we envision that important on-line certificate authorities would have large enough memories to avoid frequent disk accesses, like DNS second-level servers.

The SFS read-only protocol performs client-side name resolution, unlike the Web server which performs server-side name resolution. We measured both single-component and multi-component lookups. (For instance, `http://host/a.html` causes a single-component lookup while `http://host/a/b/c/d.html` caused a multi-component

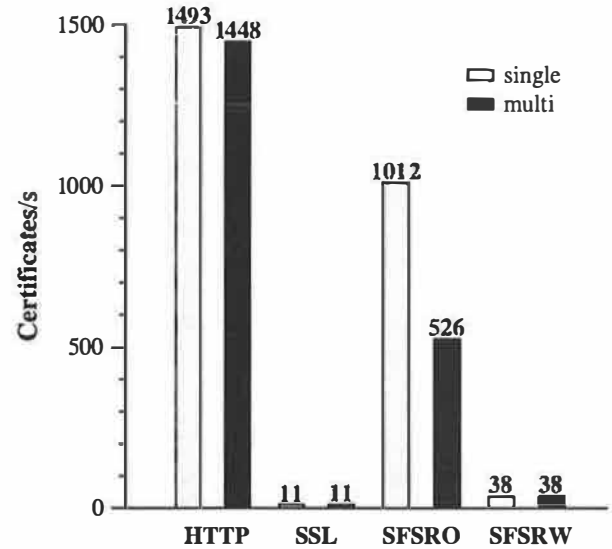


Figure 10: Maximum sustained certificate downloads per second. HTTP is an insecure Web server, SSL is a secure Web server, SFSRW is the secure SFS read-write file system, and SFSRO is the secure read-only file system. Light bars represent single-component lookups while dark bars represent multi-component lookups.

lookup.) The read-only client makes a linear number of RPCs with respect to the number of components in a lookup. On the other hand, the HTTP client makes only one HTTP request regardless of the number of components in the URL path.

The HTTP and SSL single- and multi-component tests consist of a GET `/symlink.txt` and GET `/one/two/three/symlink.txt` respectively, where `symlink.txt` contains the string `/sfs/new-york.lcs.mit.edu:bzcc5hder7cuc86kf6qswyx6yuemn w69/`. The SFSRO and SFSRW tests consist of comparable operations. We play a trace of reading a symlink that points to the above self-certifying path. The single-component SFSRO trace consists of 5 RPCs to read a symlink in the top-level directory. The multi-component trace consists of 11 RPCs to read a symlink in a directory three levels deep. The single-component SFSRW trace consists of 6 RPCs while the multi-component trace consists of 12 RPCs.

Figure 10 shows that the read-only server scales well. For single-component lookups, the SFS read-only server can process 26 times more certificate downloads than the SFS read-write server because the read-only server performs no on-line crypto-

graphic operations. The read-write server is bottlenecked by public key decryptions, which each take 24 msec. Hence, the read-write server can at best achieve 38 (1000/24) connections per second. By comparing the read-write server to the Apache Web server with SSL, we see that the read-write server is in fact quite efficient; the SSL protocol requires more and slower cryptographic operations on the server than the SFS read-only protocol.

By comparing the read-only server with an insecure Apache server, we can conclude that the read-only server is a good platform for serving read-only data to many clients; the number of connections per second is only 32% lower than that of the insecure Apache server. In fact, the performance of SFS read-only is within an order of magnitude of the performance of a DNS root server, which according to Network Solutions can sustain about 4,000 lookups per second (DNS uses UDP instead of TCP). Since the DNS root servers can support on-line name resolution for the Internet, this comparison suggests that it is reasonable to build a distributed on-line certificate authority using SFS read-only servers.

A multi-component lookup is faster with HTTP than with SFSRO. The SFSRO client must make two RPCs per component. Hence, there is a slowdown for deep directories. In practice, the impact on performance will depend on whether clients do multi-component lookups once, and then never look at the same directory again, or rather, amortize the cost of walking the file system over multiple lookups. In any situation in which a single read-only client does multiple lookups in the same directory, the client should have performance similar to the single-component case because it will cache the components along the path.

In the case of our CA benchmark, it is realistic to expect all files to reside in the root directory. Thus, this usage scenario minimizes people's true multi-component needs. On the other hand, if the root directory is huge, then SFS read-only will require a logarithmic number of round-trips for a lookup. However, SFS read-only will still outperform HTTP on a typical file system because Unix typically performs directory lookups in time linear in the number of directory entries; SFS read-only performs a lookup in logarithmic time in the number of directory entries.

7 Conclusion

The SFS read-only file system is a distributed file system that allows a high number of clients to securely access public, read-only data. The data of the file system is stored in a database, which is signed off-line with the private key of the file system. The private key of the file system does not have to be on-line, allowing it to be replicated on many untrusted machines. To allow for frequent updates, the database can be replicated incrementally. The read-only file systems pushes the cost of cryptographic operations from the server to the clients, allowing read-only servers to be simple and to support many clients. An implementation of the design in the context of the SFS global file system confirms that the read-only file system can support a large number of clients, while providing individual clients with acceptable application performance.

8 Acknowledgments

We would like to thank Chuck Blake for his help in setting up the experimental environment, Emmett Witchel for the original SFS read-only implementation, Butler Lampson for shepherding this paper, and Sameer Ajmani, the members of PDOS, and the MIT Applied Security Reading Group for their comments and suggestions.

SFS is free software available from <http://www.fs.net/> or sfs/sfs.fs.net:eu4cvv6wcnzser98yn4qjpjnn9iv6pi/.

References

- [1] Mihir Bellare and Phillip Rogaway. The exact security of digital signatures—how to sign with RSA and Rabin. In U. Maurer, editor, *Advances in Cryptology—Eurocrypt 1996*, volume 1070 of *Lecture Notes in Computer Science*, pages 399–416. Springer-Verlag, 1996.
- [2] Andrew D. Birrell, Andy Hisgen, Chuck Jerian, Timothy Mann, and Garret Swart. The Echo distributed file system. Technical Report 111, Digital Systems Research Center, Palo Alto, CA, September 1993.
- [3] B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification. RFC 1813, Network Working Group, June 1995.

- [4] Brent Callaghan and Tom Lyon. The auto-mounter. In *Proceedings of the Winter 1989 USENIX*, pages 43–51. USENIX, 1989.
- [5] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Third Symposium on Operating Systems Design and Implementation*, pages 173–186, February 1999.
- [6] Miguel Castro and Barbara Liskov. Proactive recovery in a byzantine-fault-tolerant system. Technical report, Massachusetts Institute of Technology, 1999. Submitted for publication, <http://www.pmg.lcs.mit.edu/~castro/application/recovery.pdf>.
- [7] D. Duchamp. A toolkit approach to partially disconnected operation. In *Proc. USENIX 1997 Ann. Technical Conf.*, pages 305–318. USENIX, January 1997.
- [8] D. Eastlake and C. Kaufman. Domain name system security extensions. RFC 2065, Network Working Group, January 1997.
- [9] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, April 1995.
- [10] W. Ford and M.S. Baum. *Secure electronic commerce*. Prentice Hall, 1997.
- [11] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The SSL protocol version 3.0. Internet draft (draft-freier-ssl-version3-02.txt), Network Working Group, November 1996. Work in progress.
- [12] Kevin Fu. Group sharing and random access in cryptographic storage file systems. Master's thesis, Massachusetts Institute of Technology, June 1999.
- [13] R. Gennaro and P. Rohatgi. How to sign digital streams. In *Advances in Cryptology - Crypto '97*, pages 180–197. Springer-Verlag, 1997. Lecture Notes in Computer Science Volume 1294.
- [14] Howard Gobioff, David Nagle, and Garth Gibson. Embedded security for network-attached storage. Technical Report CMU-CS-99-154, CMU, June 1999.
- [15] David Karger, Tom Leighton, Danny Lewin, and Alex Sherman. Web caching with consistent hashing. In *The eighth Word Wide Web Conference*, Toronto, Canada, May 1999.
- [16] U. Maheshwari, Radek Vingralek, and William Shapiro. How to build a trusted database system on untrusted storage. In *Fourth Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.
- [17] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 124–139, Kiawah Island, SC, 1999. ACM.
- [18] R. C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology - Crypto '87*, pages 369–378, Berlin, 1987. Springer-Verlag. Lecture Notes in Computer Science Volume 293.
- [19] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.
- [20] Pankaj Rohatgi. A compact and fast hybrid signature scheme for multicast packet authentication. In *CCS'99*, Singapore, November 1999.
- [21] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 1–15, Pacific Grove, CA, October 1991. ACM.
- [22] M. Satyanarayanan. Scalable, secure and highly available file access in a distributed workstation environment. *IEEE Computer*, pages 9–21, May 1990.
- [23] Sleepycat software. *The Berkeley Database (version 3.0.55)*. www.sleepycat.com.
- [24] R. Srinivasan. XDR: External data representation standard. RFC 1832, Network Working Group, August 1995.
- [25] TTCP. <ftp://ftp.sgi.com/sgi/src/ttcp/>.
- [26] Hugh C. Williams. A modification of the RSA public-key encryption procedure. *IEEE Transactions on Information Theory*, IT-26(6):726–729, November 1980.
- [27] C. Wong and S. Lam. Digital signatures for flows and multicasts. In *IEEE ICNP'98*, Austin, TX, October 1998.
- [28] www.rpm.org. *RPM software packaging tool*. www.rpm.org.

Overcast: Reliable Multicasting with an Overlay Network

John Jannotti David K. Gifford Kirk L. Johnson

M. Frans Kaashoek James W. O'Toole, Jr.

Cisco Systems

{jj,gifford,tuna,kaashoek,otoole}@cisco.com

Abstract

Overcast is an application-level multicasting system that can be incrementally deployed using today's Internet infrastructure. These properties stem from Overcast's implementation as an *overlay network*. An overlay network consists of a collection of nodes placed at strategic locations in an existing network fabric. These nodes implement a network abstraction on top of the network provided by the underlying *substrate* network.

Overcast provides scalable and reliable single-source multicast using a simple protocol for building efficient data distribution trees that adapt to changing network conditions. To support fast joins, Overcast implements a new protocol for efficiently tracking the global status of a changing distribution tree.

Results based on simulations confirm that Overcast provides its added functionality while performing competitively with IP Multicast. Simulations indicate that Overcast quickly builds bandwidth-efficient distribution trees that, compared to IP Multicast, provide 70%-100% of the total bandwidth possible, at a cost of somewhat less than twice the network load. In addition, Overcast adapts quickly to changes caused by the addition of new nodes or the failure of existing nodes without causing undue load on the multicast source.

1 Introduction

Overcast is motivated by real-world problems faced by content providers using the Internet today. How can bandwidth-intensive content be offered on demand? How can long-running content be offered to vast numbers of clients? Neither of these challenges are met by today's infrastructure, though for different reasons. Bandwidth-intensive content (such as 2Mbit/s video) is impractical because the bottleneck bandwidth between content providers and

consumers is considerably less than the natural consumption rate of such media. With currently available bandwidth, a 10-minute news clip might require an hour of download time. On the other hand, large-scale (thousands of simultaneous viewers) use of even moderate-bandwidth live video streams (perhaps 128Kbit/s) is precluded because network costs scale linearly with the number of consumers.

Overcast attempts to address these difficulties by combining techniques from a number of other systems. Like IP Multicast, Overcast allows data to be sent once to many destinations. Data are replicated at appropriate points in the network to minimize bandwidth requirements while reaching multiple destinations. Overcast also draws from work in caching and server replication. Overcast's multicast capabilities are used to fill caches and create server replicas throughout a network. Finally Overcast is designed as an *overlay network*, which allows Overcast to be incrementally deployed. As nodes are added to an Overcast system the system's benefits are increased, but Overcast need not be deployed universally to be effective.

An Overcast system is an overlay network consisting of a central source (which may be replicated for fault tolerance), any number of internal Overcast nodes (standard PCs with permanent storage) sprinkled throughout a network fabric, and standard HTTP clients located in the network. Using a simple tree-building protocol, Overcast organizes the internal nodes into a distribution tree rooted at the source. The tree-building protocol adapts to changes in the conditions of the underlying network fabric. Using this distribution tree, Overcast provides large-scale, reliable multicast groups, especially suited for on-demand and live data delivery. Overcast allows unmodified HTTP clients to join these multicast groups.

Overcast permits the archival of content sent to multicast groups. Clients may specify a starting point

when joining an archived group, such as the beginning of the content. This feature allows a client to “catch up” on live content by tuning back ten minutes into a stream, for instance. In practice, the nature of a multicast group will most often determine the way it is accessed. A group containing stock quotes will likely be accessed live. A group containing a software package will likely be accessed from start to finish; “live” would have no meaning for such a group. Similarly, high-bandwidth content can not be distributed live when the bottleneck bandwidth from client to server is too small. Such content will always be accessed relative to its start.

We have implemented Overcast and used it to create a data distribution system for businesses. Most current users distribute high quality video that clients access on demand. These businesses operate geographically distributed offices and need to distribute video to their employees. Before using Overcast, they met this need with low resolution Web-accessible video or by physically reproducing and mailing VHS tapes. Overcast allows these users to distribute high-resolution video over the Internet. Because high quality videos are large (Approximately 1 Gbyte for a 30 minute MPEG-2 video), it is important that the videos are efficiently distributed and available from a node with high bandwidth to the client. To a lesser extent, Overcast is also being used to broadcast live streams. Existing Overcast networks typically contain tens of nodes and are scheduled to grow to hundreds of nodes.

The main challenge in Overcast is the design and implementation of protocols that can build efficient, adaptive distribution trees without knowing the details of the substrate network topology. The substrate network’s abstraction provides the appearance of direct connectivity between all Overcast nodes. Our goal is to build distribution trees that maximize each node’s bandwidth from the source and utilize the substrate network topology efficiently. For example, the Overcast protocols should attempt to avoid sending data multiple times over the same physical link. Furthermore, Overcast should respond to transient failures or congestion in the substrate network.

Consider the simple network depicted in Figure 1. The network substrate consists of a root node (R), two Overcast nodes (O), a router, and a number of links. The links are labeled with bandwidth in Mbit/s. There are three ways of organizing the root and the Overcast nodes into a distribution tree. The organization shown optimizes bandwidth by using

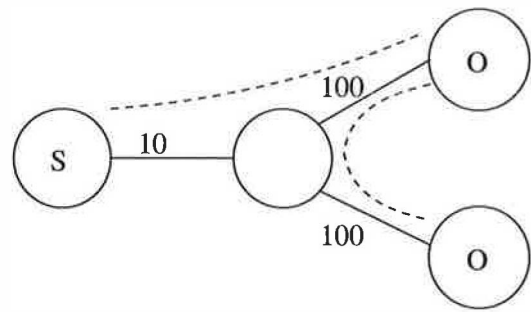


Figure 1: An example network and Overcast topology. The straight lines are the links in the substrate network. These links are labeled with bandwidth in Mbit/s. The curved lines represent connections in the Overlay network. **S** represents the source, **O** represents two Overcast nodes.

the constrained link only once.

The contributions of this paper are:

- A novel use of overlay networks. We describe how reliable, highly-scalable, application-level multicast can be provided by adding nodes that have permanent storage to the existing network fabric.
- A simple protocol for forming efficient and scalable distribution trees that adapt to changes in the conditions of the substrate network without requiring router support.
- A novel protocol for maintaining global status at the root of a changing distribution tree. This state allows clients to join an Overcast group quickly while maintaining scalability.
- Results from simulations that show Overcast is efficient. Overcast can scale to a large number of nodes; its efficiency approaches router-based systems; it quickly adjusts to configuration changes; and a root can track the status of an Overcast network in a scalable manner.

Section 2 details Overcast’s relation to prior work. Overcast’s general structure is examined in Section 3, first by describing overlay networks in general, then providing the details of Overcast. Section 4 describes the operation of the Overcast network performing reliable application-level multicast. Finally, Section 5 examines Overcast’s ability to build a bandwidth-efficient overlay network for multicasting and to adapt efficiently to changing network conditions.

2 Related Work

Overcast seeks to marry the bandwidth savings of an IP Multicast distribution tree with the reliability and simplicity of store-and-forward operation using reliable communication between nodes. Overcast builds on research in IP multicast, content distribution (caching, replication, and content routing), and overlay networks. We discuss each in turn.

IP Multicast IP Multicast [11] is designed to provide efficient group communication as a low level network primitive. Overcast has a number of advantages over IP Multicast. First, as it requires no router support, it can be deployed incrementally on existing networks. Second, Overcast provides bandwidth savings both when multiple clients view content simultaneously and when multiple clients view content at different times. Third, while reliable multicast is the subject of much research [19, 20], problems remain when various links in the distribution tree have widely different bandwidths. A common strategy in such situations is to decrease the fidelity of content over lower bandwidth links. Although such a strategy has merit when content *must* be delivered live, Overcast also supports content types that require bit-for-bit integrity, such as software.

Express [15] is a single-source multicasting system that addresses some of IP Multicast's deficits. Express alleviates difficulties relating to IP Multicast's small address space, susceptibility to denial of service attacks, and billing difficulties which may lie at the root of IP Multicast's lack of deployment on commercial networks. In these three respects Overcast bears a great deal of similarity to Express. Overcast differs mainly by stressing deployability and flexibility. Overcast does not require router modifications, simplifying adoption and increasing flexibility. Although Overcast provides a useful range of functionality, we recognize that there needs for which Overcast may not be suited. Express standardizes a single model in the router which works to lock out applications with different needs.

Content Distribution Systems Others have advocated distributing content servers in the network fabric, from initial proposals [10] to larger projects, such as Adaptive Caching [26], Push Caching [14], Harvest [8], Dynamic Hierarchical Caching [7], Speculative Data Dissemination [6], and Application-Level Replication [4]. Overcast extends this previous work by building an overlay network using a self-organizing algorithm. This algorithm, operating continuously, not only eliminates

the need for manually determined topology information when the overlay network is created, but also reacts transparently to the addition or removal of nodes in the running system. Initialization, expansion, and fault tolerance are unified.

A number of service providers (e.g., Adero, Akamai, and Digital Island) operate content distribution networks, but in-depth information describing their internals is not public information. FastForward's product is described below as an example of an overlay network.

Overlay Networks A number of research groups and service providers are investigating services based on overlay networks. In particular, many of these services, like Overcast, exist to provide some form of multicast or content distribution. These include End System Multicast [16], Yoid [13] (formerly Yallcast), X-bone [24], RMX [9], FastForward [1], and PRISM [5]. All share the goal of providing the benefits of IP multicast without requiring direct router support or the presence of a physical broadcast medium. However, except Yoid, these approaches do not exploit the presence of permanent storage in the network fabric.

End System Multicast is an overlay network that provides small-scale multicast groups for teleconferencing applications; as a result the End System Multicast protocol (Narada) is designed for multi-source multicast. The Overcast protocols differ from Narada in order to support large-scale multicast groups.

Yoid is a generic architecture for overlay networks with a number of new protocols, which are in development. The most striking difference between Yoid and Overcast is in approach. Yoid strives to be a general purpose overlay network and content distribution toolkit, addressing applications as diverse as netnews, streaming broadcasts, and bulk email distribution. While these goals are laudable, we believe that because Overcast is more focused on providing single-source multicast our protocols are simpler to understand and implement. Nonetheless, there remains a great deal of similarity between Overcast and Yoid, including url-like group naming, the use of disk space to "time-shift" multicast distribution, and automatic tree configuration.

X-bone is also a general-purpose overlay network that can support many different network services. The overlay networks formed by X-bone are meshes, which are statically configured.

RMX focuses on real-time reliable multicast. As such, its focus is on reconciling the heterogeneous capabilities and network connections of various clients with the need for reliability. Therefore their work focuses on *semantic* rather than *data* reliability. For instance, RMX can be used to change high resolution images into progressive JPEGs before transmittal to underprovisioned clients. Our work is less concerned with interactive response times. Overcast is designed for content that clients are interested in only at full fidelity, even if it means that the content does not become available to all clients at the same time.

FastForward Networks produces a system sharing many properties with RMX. Like RMX, FastForward focuses on real-time operation and includes provisions for intelligently decreasing the bandwidth requirements of rich media for low-bandwidth clients. Beyond this, FastForward's product differs from Overcast in that its distribution topology is statically configured by design. Within this statically configured topology, the product can pick dynamic routes. In this way FastForward allows experts to configure the topology for better performance and predictability while allowing for a limited degree of dynamism. Overcast's design seeks to minimize human intervention to allow its overlay networks to scale to thousands of nodes. Similarly, FastForward achieves fault tolerance by statically configuring distribution topologies to avoid single points of failure, while Overcast seeks to dynamically reconfigure its overlay in response to failures.

PRISM is an architecture for distributing streaming media over IP. Its architecture bears some similarity to Overcast, but their work appears focused on the naming of content and the design of interior nodes of the system. PRISM's high level design includes an overlay based content distribution mechanism, but it is assumed that such a system can be "plugged in" to the rest of PRISM. Overcast could provide that mechanism.

Active Services Active Services [2] is a framework for implementing services at the application-level throughout the fabric of the network. In that sense, there is a strong similarity in mindset between our works. However, Active Services must contend with the difficulty of sharing the resources of a single computer among multiple services, a difficulty we avoid by using dedicated nodes. Perhaps because of this challenge, Active Service applications have focused on real-time multimedia streaming, an application with transient resource needs. Our ap-

plication uses large amounts of disk space for long periods of time which is problematic in a shared environment.

Our observation is that one-time hardware costs do not drive the total costs of systems on the scale that we propose. Total cost is dominated by bandwidth, maintenance, and continual hardware obsolescence. Therefore Overcast seeks to minimize the use of bandwidth, cut maintenance costs by simplifying node deployment, and avoid obsolescence by structuring the system to allow older nodes to continue to contribute to the total efficiency of the overlay network.

Active Networks One may view overlay networks as an alternative implementation of active networks [23]. In active networks, new protocols and application-code can dynamically be downloaded into routers, allowing for rapid innovation of network services. Overcast avoids some of the hard problems of active networks by focusing on a single application; it does not have to address the problems created by dynamic downloading of code and sharing resources among multiple competing applications. Furthermore, since Overcast requires no changes to existing routers, it is easier to deploy. The main challenge for Overcast is to be competitive with solutions that are directly implemented on the network level.

3 The Overcast Network

This section describes the overlay network created by the Overcast system. First, we argue the benefits and drawbacks of using an overlay network. After concluding that an overlay network is appropriate for the task at hand, we explore the particular design of an overlay network to meet Overcast's demands. To do so, we examine the key design requirement of the Overcast network—single source distribution of bandwidth-intensive media on today's Internet infrastructure. Finally we illustrate the use of Overcast with an example.

3.1 Why overlay?

Overcast was designed to meet the needs of content providers on the Internet. This goal led us to an overlay network design. To understand why we chose an overlay network, we consider the benefits and drawbacks of overlays.

An overlay network provides advantages over both centrally located solutions and systems that advocate running code in every router. An overlay network is:

Incrementally Deployable An overlay network requires *no* changes to the existing Internet infrastructure, only additional servers. As nodes are added to an overlay network, it becomes possible to control the paths of data in the substrate network with ever greater precision.

Adaptable Although an overlay network abstraction constrains packets to flow over a constrained set of links, that set of links is constantly being optimized over metrics *that matter to the application*. For instance, the overlay nodes may optimize latency at the expense of bandwidth. The Detour Project [21] has discovered that there are often routes between two nodes with less latency than the routes offered by today's IP infrastructure. Overlay networks can find and take advantage of such routes.

Robust By virtue of the increased control and the adaptable nature of overlay networks, an overlay network can be *more* robust than the substrate fabric. For instance, with a sufficient number of nodes deployed, an overlay network may be able to guarantee that it is able to route between any two nodes in two independent ways. While a robust substrate network can be expected to repair faults eventually, such an overlay network might be able to route around faults immediately.

Customizable Overlay nodes may be multi-purpose computers, easily outfitted with whatever equipment makes sense. For example, Overcast makes extensive use of disk space. This allows Overcast to provide bandwidth savings even when content is not consumed simultaneously in different parts of the network.

Standard An overlay network can be built on the least common denominator network services of the substrate network. This ensures that overlay traffic will be treated as well as any other. For example, Overcast uses TCP (in particular, HTTP over port 80) for reliable transport. TCP is simple, well understood, network friendly, and standard. Alternatives, such as a "home grown" UDP protocol with retransmissions, are less attractive by all these measures. For better or for worse, creativity in reliable transport is a losing battle on the Internet today.

On the other hand, building an overlay network faces a number of interesting challenges. An overlay network must address:

Management complexity The manager of an overlay network is physically far removed from the machines being managed. Routine maintenance must either be unnecessary or possible from afar, using tools that do not scale in complexity with the size of the network. Physical maintenance must be minimized and be possible by untrained personnel.

The real world In the real world, IP *does not* provide universal connectivity. A large portion of the Internet lies behind firewalls. A significant and growing share of hosts are behind Network Address Translators (NATs), and proxies. Dealing with these practical issues is tedious, but crucial to adoption.

Inefficiency An overlay *can not* be as efficient as code running in every router. However, our observation is that when an overlay network is small, the inefficiency, measured in absolute terms, will be small as well — and as the overlay network grows, its efficiency can approach the efficiency of router based services.

Information loss Because the overlay network is built on top of a network infrastructure (IP) that offers nearly complete connectivity (limited only by firewalls, NATs, and proxies), we expend considerable effort deducing the topology of the substrate network.

The first two of these problems can be addressed and nearly eliminated by careful design. To address management complexity, management of the entire overlay network can be concentrated at a single site. The key to a centralized-administration design is guaranteeing that newly installed nodes can boot and obtain network connectivity without intervention. Once that is accomplished, further instructions may be read from the central management server.

Firewalls, NATs and HTTP proxies complicate Overcast's operation in a number of ways. Firewalls force Overcast to open all connections "upstream" and to communicate using HTTP on port 80. This allows an Overcast network to extend exactly to those portions of the Internet that allow web browsing. NATs are devices used to multiplex a small set of IP addresses (often exactly one) over a number of clients. The clients are configured to use the NAT as their default router. At the NAT, TCP connections are rewritten to use one of the small number of IP addresses managed by the NAT. TCP port numbers allow the NAT to demultiplex return

packets back to the correct client. The complication for Overcast is that client IP addresses are obscured. All Overcast nodes behind the NAT appear to have the same IP address. HTTP proxies have the same effect.

Although private IP addresses are never directly used by external Overcast nodes, there are times when an external node must correctly report the private IP address of another node. For example, an external node may have internal children. During tree building a node must report its children's addresses so that they may be measured for suitability as parents themselves. Only the private address is suitable for such purposes. To alleviate this complication all Overcast messages contain the sender's IP address *in the payload* of the message.

The final two disadvantages are not so easily dismissed. They represent the true tradeoff between overlay networks and ubiquitous router based software. For Overcast, the goal of instant deployment is important enough to sacrifice some measure of efficiency. However, the amount of inefficiency introduced is a key metric by which Overcast should be judged.

3.2 Single-Source Multicast

Overcast is a single-source multicast system. This contrasts with IP Multicast which allows any member of a multicast group to send packets to all other members of the group. Beyond the fact that this closely models our intended application domain, there are a number of reasons to pursue this particular refinement to the IP Multicast model.

Simplicity Both conceptually and in implementation, a single-source system is simpler than an any-source model. For example, a single-source provides an obvious rendezvous point for group joins.

Optimization It is difficult to optimize the structure of the overlay network without intimate knowledge of the substrate network topology. This only becomes harder if the structure must be optimized for all paths [16].

Address space Single-source multicast groups provide a convenient alternative to the limited IP Multicast address space. The namespace can be partitioned by first naming the source, then allowing further subdivision of the source's choosing. In contrast, IP Multicast's address space is flat, limited,

and without obvious administration to avoid collisions amongst new groups.

On the other hand, a single-source model clearly offers reduced functionality compared to a model that allows any group member to multicast. As such, Overcast is not appropriate for applications that require extensive use of such a model. However, many applications which appear to need multi-source multicast, such as a distributed lecture allowing questions from the class, do not. In such an application, only one "non-root" sender is active at any particular time. It would be a simple matter for the sender to unicast to the root, which would then perform the true multicast on the behalf of the sender. A number of projects [15, 17, 22] have used or advocated such an approach.

3.3 Bandwidth Optimization

Overcast is designed for distribution from a single source. As such, small latencies are expected to be of less importance to its users than increased bandwidth. Extremely low latencies are only important for applications that are inherently two-way, such as video conferencing. Overcast is designed with the assumption that broadcasting "live" video on the Internet may actually mean broadcasting with a ten to fifteen second delay.

Overcast distribution trees are built with the sole goal of creating high bandwidth channels from the source to all nodes. Although Overcast makes no guarantees that the topologies created are optimal, our simulations show that they perform quite well. The exact method by which high-bandwidth distribution trees are created and maintained is described in Section 4.2.

3.4 Deployment

An important goal for Overcast is to be deployable on today's Internet infrastructure. This motivates not only the use of an overlay network, but many of its details. In particular, deployment must require little or no human intervention, costs per node should be minimized, and unmodified HTTP clients must be able to join multicast groups in the Overcast network.

To help ease the human costs of deployment, nodes in the Overcast network configure themselves in an

adaptive distributed tree with a single root. No human intervention is required to build efficient distribution trees, and nodes can be a part of multiple distribution trees.

Overcast's implementation on commodity PCs running Linux further eases deployment. Development is speeded by the familiar programming environment, and hardware costs are minimized by continually tracking the best price/performance ratio available in off-the-shelf hardware. The exact hardware configuration we have deployed has changed many times in the year or so that we have deployed Overcast nodes.

The final consumers of content from an Overcast network are HTTP clients. The Overcast protocols are carefully designed so that unmodified Web browsers can become members of a multicast group. In Overcast, a multicast group is represented as an HTTP URL: the hostname portion names the root of an Overcast network and the path represents a particular group on the network. All groups with the same root share a single distribution tree.

Using URLs as a namespace for Overcast groups has three advantages. First, URLs offer a hierarchical namespace, addressing the scarcity of multicast group names in traditional IP Multicast. Second, URLs and the means to access them are an existing standard. By delivering data over a simple HTTP connection, Overcast is able to bring multicasting to unmodified applications. Third, a URL's richer structure allows for simple expression of the increased power of Overcast over tradition multicast. For example, a group suffix of `start=10s` may be defined to mean "begin the content stream 10 seconds from the beginning."

3.5 Example usage

We have used Overcast to build a content-distribution application for high-quality video and live streams. The application is built out of a publishing station (called a *studio*) and nodes (called *appliances*). Appliances are installed at strategic locations in their network. The appliances boot, contact their studio, and self-organize into a distribution tree, as described below. No local administration is required.

The studio stores content and schedules it for delivery to the appliances. Typically, once the content is delivered, the publisher at the studio generates

a web page announcing the availability of the content. When a user clicks on the URL for published content, Overcast redirects the request to a nearby appliance and the appliance serves the content. If the content is video, no special streaming software is needed. The user can watch the video over standard protocols and a standard MPEG player, which is supplied with most browsers.

An administrator at the studio can control the overlay network from a central point. She can view the status of the network (*e.g.*, which appliances are up), collect statistics, control bandwidth consumption, etc.

Using this system, bulk data can be distributed efficiently, even if the network between the appliances and the studio consists of low-bandwidth or intermittent links. Given the relative prices of disk space and network bandwidth, this solution is far less expensive than upgrading all network links between the studio and every client.

4 Protocols

The previous section described the structure and properties of the Overcast overlay network. This section describes how it functions: the initialization of individual nodes, the construction of the distribution hierarchy, and the automatic maintenance of the network. In particular, we describe the "tree" protocol to build distribution trees and the "up/down" protocol to maintain the global state of the Overcast network efficiently. We close by describing how clients (web browsers) join a group and how reliable multicasting to clients is performed.

4.1 Initialization

When a node is first plugged in or moved to a new location it automatically initializes itself and contacts the appropriate Overcast root(s). The first step in the initialization process is to determine an IP address and gateway address that the node can use for general IP connectivity. If there is a local DHCP server then the node can obtain IP configuration directly data using the DHCP protocol [12]. If DHCP is unavailable, a utility program can be used from a nearby workstation for manual configuration.

Once the node has an IP configuration it contacts a global, well-known registry, sending along its unique

serial number. Based on a node's serial number, the registry provides a list of the Overcast networks the node should join, an optional permanent IP configuration, the network areas it should serve, and the access controls it should implement. If a node is intended to become part of a particular content distribution network, the configuration data returned will be highly specific. Otherwise, default values will be returned and the networks to which a node will join can be controlled using a web-based GUI.

4.2 The Tree Building Protocol

Self-organization of appliances into an efficient, robust distribution tree is the key to efficient operation in Overcast. Once a node initializes, it begins a process of self-organization with other nodes of the same Overcast network. The nodes cooperatively build an overlay network in the form of a distribution tree with the root node at its source. This section describes the tree-building protocol.

As described earlier, the virtual links of the overlay network are the only paths on which data is exchanged. Therefore the choice of distribution tree can have a significant impact on the aggregate communication behavior of the overlay network. By carefully building a distribution tree, the network utilization of content distribution can be significantly reduced. Overcast stresses bandwidth over other conceivable metrics, such as latency, because of its expected applications. Overcast is not intended for interactive applications, therefore optimizing a path to shave small latencies at the expense of total throughput would be a mistake. On the other hand, Overcast's architecture as an overlay network allows this decision to be revisited. For instance, it may be decided that trees should have a fixed maximum depth to limit buffering delays.

The goal of Overcast's tree algorithm is to maximize bandwidth to the root for all nodes. At a high level the algorithm proceeds by placing a new node as far away from the root as possible without sacrificing bandwidth to the root. This approach leads to "deep" distribution trees in which the nodes nonetheless observe no worse bandwidth than obtaining the content directly from the root. By choosing a parent that is nearby in the network, the distribution tree will form along the lines of the substrate network topology.

The tree protocol begins when a newly initialized node contacts the root of an Overcast group. The

root thereby becomes the *current* node. Next, the new node begins a series of rounds in which it will attempt to locate itself further away from the root without sacrificing bandwidth back to the root. In each round the new node considers its bandwidth to *current* as well as the bandwidth to *current through each of current's children*. If the bandwidth through any of the children is about as high as the direct bandwidth to *current*, then one of these children becomes *current* and a new round commences. In the case of multiple suitable children, the child closest (in terms of network hops) to the searching node is chosen. If no child is suitable, the search for a parent ends with *current*.

To approximate the bandwidth that will be observed when moving data, the tree protocol measures the download time of 10 Kbytes. This measurement includes all the costs of serving actual content. We have observed that this approach to measuring bandwidth gives us better results than approaches based on low-level bandwidth measurements such as using ping. On the other hand, we recognize that a 10 Kbyte message is too short to accurately reflect the bandwidth of "long fat pipes". We plan to move to a technique that uses progressively larger measurements until a steady state is observed.

When the measured bandwidths to two nodes are within 10% of each other, we consider the nodes equally good and select the node that is closest, as reported by *traceroute*. This avoids frequent topology changes between two nearly equal paths, as well as decreasing the total number of network links used by the system.

A node periodically reevaluates its position in the tree by measuring the bandwidth to its current siblings (an up-to-date list is obtained from the parent), parent, and grandparent. Just as in the initial building phase, a node will relocate below its siblings if that does not decrease its bandwidth back to the root. The node checks bandwidth directly to the grandparent as a way of testing its previous decision to locate under its current parent. If necessary the node moves back up in the hierarchy to become a sibling of its parent. As a result, nodes constantly reevaluate their position in the tree and an Overcast network is inherently tolerant of non-root node failures. If a node goes off-line for some reason, any nodes that were below it in the tree will reconnect themselves to the rest of the routing hierarchy. When a node detects that its parent is unreachable, it will simply relocate beneath its

grandparent. If its grandparent is also unreachable the node will continue to move up its ancestry until it finds a live node. The ancestor list also allows cycles to be avoided as nodes asynchronously choose new parents. A node simply refuses to become the parent of a node it believes to be its own ancestor. A node that chooses such a node will be forced to rechoose.

While there is extensive literature on faster fail-over algorithms, we have not yet found a need to optimize beyond the strategy outlined above. It is important to remember that the nodes participating in this protocol are dedicated machines that are less prone to failure than desktop computers. If this becomes an issue, we have considered extending the tree building algorithm to maintain backup parents (excluding a node's own ancestry from consideration) or an entire backup tree.

By periodically remeasuring network performance, the overlay network can adapt to network conditions that manifest themselves at time scales larger than the frequency at which the distribution tree reorganizes. For example, a tree that is optimized for bandwidth efficient content delivery during the day may be significantly suboptimal during the overnight hours (when network congestion is typically lower). The ability of the tree protocol to automatically adapt to these kinds of changing network conditions provides an important advantage over simpler, statically configured content distribution schemes.

4.3 The Up/Down Protocol

To allow web clients to join a group quickly, the Overcast network must track the status of the Overcast nodes. It may also be important to report statistical information back to the root, so that content providers might learn, for instance, how often certain content is being viewed. This section describes a protocol for efficient exchange of information in a tree of network nodes to provide the root of the tree with information from nodes throughout the network. For our needs, this protocol must scale sublinearly in terms of network usage at the root, but may scale linearly in terms of space (all with respect to the number of Overcast nodes). This is a simple result of the relative requirements of a client for these two resources and the cost of those resources. Overcast might store (conservatively) a few hundred bytes about each Overcast node, but even in a group of millions of nodes, total RAM cost for the root would be under \$1,000.

We call this protocol the “up/down” protocol because our current system uses it mainly to keep track of what nodes are up and what nodes are down. However, arbitrary information in either of two large classes may be propagated to the root. In particular, if the information either changes slowly (*e.g.*, up/down status of nodes), or the information can be combined efficiently from multiple children into a single description (*e.g.*, group membership counts), it can be propagated to the root. Rapidly changing information that can not be aggregated during propagation would overwhelm the root's bandwidth capacity.

Each node in the network, including the root node, maintains a table of information about all nodes lower than itself in the hierarchy and a log of all changes to the table. Therefore the root node's table contains up-to-date information for all nodes in the hierarchy. The table is stored on disk and cached in the memory of a node.

The basis of the protocol is that each node periodically checks in with the node directly above it in the tree. If a child fails to contact its parent within a preset interval, the parent will assume the child and all its descendants have “died”. That is, either the node has failed, an intervening link has failed, or the child has simply changed parents. In any case, the parent node marks the child and its descendants “dead” in its table. Parents never initiate contact with descendants. This is a byproduct of a design that is intended to cross firewalls easily. All node failures must be detected by a failure to check in, rather than active probing.

During these periodic check-ins, a node reports new information that it has observed or been informed of since it last checked in. This includes:

- “Death certificates” - Children that have missed their expected report time.
- “Birth certificates” - Nodes that have become children of the reporting node.
- Changes to the reporting node's “extra information.”
- Certificates or changes that have been propagated to the node from its own children since its last checkin.

This simple protocol exhibits a race condition when a node chooses a new parent. The moving node's

former parent propagates a death certificate up the hierarchy, while at nearly the same time the new parent begins propagating a birth certificate up the tree. If the birth certificate arrives at the root first, when the death certificate arrives the root will believe that the node has failed. This inaccuracy will remain indefinitely since a new birth certificate will only be sent in response to a change in the hierarchy that may not occur for an arbitrary period of time.

To alleviate this problem, a node maintains a sequence number indicating of how many times it has changed parents. All changes involving a node are tagged with that number. A node ignores changes that are reported to it about a node if it has already seen a change with a higher sequence number. For instance, a node may have changed parents 17 times. When it changes again, its former parent will propagate a death certificate annotated with 17. However, its new parent will propagate a birth certificate annotated with 18. If the birth certificate arrives first, the death certificate will be ignored since it is older.

An important optimization to the up/down protocol avoids large sets of birth certificates from arriving at the root in response to a node with many descendants choosing a new parent. Normally, when a node moves to a new parent, a birth certificate must be sent out for each of its descendants to its new parent. This maintains the invariant that a node knows the parent of all its descendants. Keep in mind that a birth certificate is not only a record that a node exists, but that it has a certain parent.

Although this large set of updates is required, it is usually unnecessary for these updates to continue far up the hierarchy. For example, when a node relocates beneath a sibling, the sibling must learn about all of the node's descendants, but when the sibling, in turn, passes these certificates to the original parent, the original parent notices that they do not represent a change and quashes the certificate from further propagation.

Using the up/down protocol, the root of the hierarchy will receive timely updates about changes to the network. The freshness of the information can be tuned by varying the length of time between check-ins. Shorter periods between updates guarantee that information will make its way to the root more quickly. Regardless of the update frequency, bandwidth requirements at the root will be proportional to the number of changes in the hierarchy rather than the size of the hierarchy itself.

4.4 Replicating the root

In Overcast, there appears to be the potential for significant scalability and reliability problems at the root. The up/down protocol works to alleviate the scalability difficulties in maintaining global state about the distribution tree, but the root is still responsible for handling all join requests from all HTTP clients. The root handles such requests by redirection, which is far less resource intensive than actually delivering the requested content. Nonetheless, the possibility of overload remains for particularly popular groups. The root is also a single point of failure.

To address this, overcast uses a standard technique used by many popular websites. The DNS name of the root resolves to any number of replicated roots in round-robin fashion. The database used to perform redirections is replicated to all such roots. In addition, IP address takeover may be used for immediate failover, since DNS caching may cause clients to continue to contact a failed replica. This simple, standard technique works well for this purpose because handling joins from HTTP clients is a read-only operation that lends well to distribution over numerous replicas.

There remains, however, a single point of failure for the up/down protocol. The functionality of the root in the up/down protocol cannot be distributed so easily because its purpose is to maintain changing state. However the up/down protocol has the useful property that all nodes maintain state for nodes below them in the distribution tree. Therefore, a convenient technique to address fault tolerance is to specially construct the top of the hierarchy.

Starting with the root, some number of nodes are configured linearly, that is, each has only one child. In this way all other overcast nodes lie below these top nodes. Figure 2 shows a distribution tree in which the top three nodes are arranged linearly. Each of these nodes has enough information to act as the root of the up/down protocol in case of a failure. This technique has the drawback of increasing the latency of content distribution unless special-case code skips the extra roots during distribution. If latency were important to Overcast this would be an important, but simple, optimization.

“Linear roots” work well with the need for replication to address scalability, as mentioned above. The set of linear nodes has all the information needed to

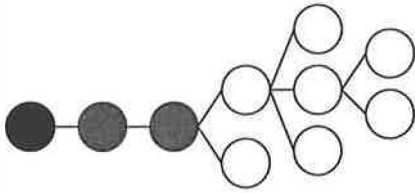


Figure 2: A specially configured distribution topology that allows either of the grey nodes to quickly stand in as the root (black) node. All filled nodes have complete status information about the unfilled nodes.

perform Overcast joins, therefore these nodes are perfect candidates to be used in the DNS round-robin approach to scalability. By choosing these nodes, no further replication is necessary.

4.5 Joining a multicast group

To join a multicast group, a Web client issues an HTTP GET request with the URL for a group. The hostname of the URL names the root node(s). The root uses the pathname of the URL, the location of the client, and its database of the current status of the Overcast nodes to decide where to connect the client to the multicast tree. Because status information is constantly propagated to the root, a decision may be made quickly without further network traffic, enabling fast joins.

Joining a group consists of selecting the best server and redirecting the client to that server. The details of the server selection algorithm are beyond the scope of this paper as considerable previous work [3, 18] exists in this area. Furthermore, Overcast's particular choices are constrained considerably by a desire to avoid changes at the client. Without such a constraint simpler choices could have been made, such as allowing clients to participate directly in the Overcast tree building protocol.

Although we do not discuss server selection here, a number of Overcast's details exist to support this important functionality, however it may actually be implemented. A centralized root performing redirections is convenient for an approach involving large tables containing collected Internet topology data. The up/down algorithm allows for redirections to nodes that are known to be functioning.

4.6 Multicasting with Overcast

We refer to reliable multicasting on an overcast network as "overcasting". Overcasting proceeds along

the distribution tree built by the tree protocol. Data is moved between parent and child using TCP streams. If a node has four children, four separate connections are used. The content may be pipelined through several generations in the tree. A large file or a long-running live stream may be in transit over tens of different TCP streams at a single moment, in several layers of the distribution hierarchy.

If a failure occurs during an overcast, the distribution tree will rebuild itself as described above. After rebuilding the tree, the overcast resumes for on-demand distributions where it left off. In order to do so, each node keeps a log of the data it has received so far. After recovery, a node inspects the log and restarts all overcasts in progress.

Live content on the Internet today is typically buffered before playback. This compensates for momentary glitches in network throughput. Overcast can take advantage of this buffering to mask the failure of a node being used to Overcast data. As long as the failure occurs in a node that is not at the edge of the Overcast network, an HTTP client need not ever become aware that the path of data from the root has been changed in the face of failure.

5 Evaluation

In this section, the protocols presented above are evaluated by simulation. Although we have deployed Overcast in the real world, we have not yet deployed on a sufficiently large network to run the experiments we have simulated.

To evaluate the protocols, an overlay network is simulated with increasing numbers of overcast nodes while keeping the total number of network nodes constant. Overcast should build better trees as more nodes are deployed, but protocol overhead may grow.

We use the Georgia Tech Internetwork Topology Models [25] (GT-ITM) to generate the network topologies used in our simulations. We use the "transit-stub" model to obtain graphs that more closely resemble the Internet than a pure random construction. GT-ITM generates a transit-stub graph in stages, first a number of random backbones (transit domains), then the random structure of each back-bone, then random "stub" graphs are attached to each node in the backbones.

We use this model to construct five different 600 node graphs. Each graph is made up of three transit domains. These domains are guaranteed to be

connected. Each transit domain consists of an average of eight stub networks. The stub networks contain edges amongst themselves with a probability of 0.5. Each stub network consists of an average of 25 nodes, in which nodes are once again connected with a probability of 0.5. These parameters are from the sample graphs in the GT-ITM distribution; we are unaware of any published work that describes parameters that might better model common Internet topologies.

We extended the graphs generated by GT-ITM with bandwidth information. Links internal to the transit domains were assigned a bandwidth of 45Mbits/s, edges connecting stub networks to the transit domains were assigned 1.5Mbits/s, finally, in the local stub domain, edges were assigned 100Mbit/s. These reflect commonly used network technology: T3s, T1s, and Fast Ethernet. All measurements are averages over the five generated topologies.

Empirical measurements from actual Overcast nodes show that a single Overcast node can easily support twenty clients watching MPEG-1 videos, though the exact number is greatly dependent on the bandwidth requirements of the content. Thus with a network of 600 overcast nodes, we are simulating multicast groups of perhaps 12,000 members.

5.1 Tree protocol

The efficiency of Overcast depends on the positioning of Overcast nodes. In our first experiments, we compare two different approaches to choosing positions. The first approach, labelled “Backbone”, preferentially chooses transit nodes to contain Overcast nodes. Once all transit nodes are Overcast nodes, additional nodes are chosen at random. This approach corresponds to a scenario in which the owner of the Overcast nodes places them strategically in the network. In the second, labelled “Random”, we select all Overcast nodes at random. This approach corresponds to a scenario in which the owner of Overcast nodes does not pay attention to where the nodes are placed.

The goal of Overcast’s tree-building protocol is to optimize the bottleneck bandwidth available back to the root for all nodes. The goal is to provide each node with the same bandwidth to the root that the node would have in an idle network. Figure 3 compares the sum of all nodes’ bandwidths back to the root in Overcast networks of various sizes to

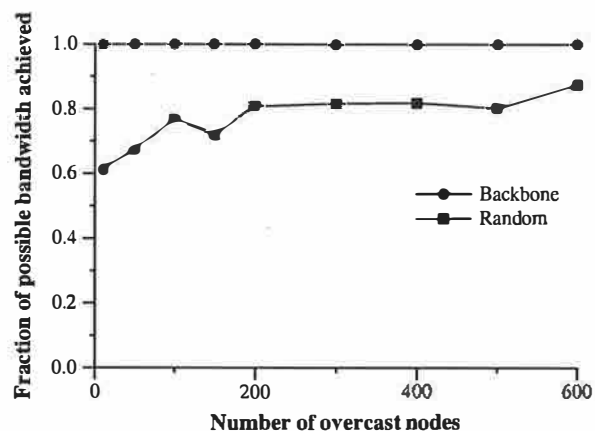


Figure 3: Fraction of potential bandwidth provided by Overcast.

the sum of all nodes’ bandwidths back to the root in an optimal distribution tree using router-based software. This indicates how well Overcast performs compared to IP Multicast.

The main observation is that, as expected, the backbone strategy for placing Overcast nodes is more effective than the random strategy, but the results of random placement are encouraging nonetheless. Even a small number of deployed Overcast nodes, positioned at random, provide approximately 70%-80% of the total possible bandwidth.

It is extremely encouraging that, when using the backbone approach, no node receives less bandwidth under Overcast than it would receive from IP Multicast. However some enthusiasm must be withheld, because a simulation artifact has been left in these numbers to illustrate a point.

Notice that the backbone approach and the random approach differ in effectiveness even when all 600 nodes of the network are Overcast nodes. In this case the same nodes are participating in the protocol, but better trees are built using the backbone approach. This illustrates that the trees created by the tree-building protocol are not unique. The backbone approach fares better by this metric because in our simulations backbone nodes were turned on first. This allowed backbone nodes to preferentially form the “top” of the tree. This indicates that in future work it may be beneficial to extend the tree-building protocol to accept hints that mark certain nodes as “backbone” nodes. These nodes would preferentially form the core of the distribution tree.

Overcast appears to perform quite well for its intended goal of optimizing available bandwidth, but

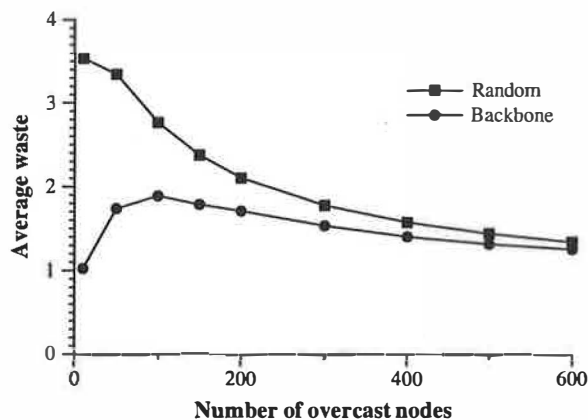


Figure 4: Ratio of the number of times a packet must “hit the wire” to be propagated through an Overcast network to a lower bound estimate of the same measure for IP Multicast.

it is reasonable to wonder what costs are associated with this performance.

To explore this question we measure the network load imposed by Overcast. We define network load to be the number of times that a particular piece of data must traverse a network link to reach all Overcast nodes. In order to compare to IP Multicast Figure 4 plots the ratio of the network load imposed by Overcast to a lower bound estimate of IP Multicast’s network load. For a given set of nodes, we assume that IP Multicast would require exactly one less link than the number of nodes. This assumes that all nodes are one hop away from another node, which is unlikely to be true in sparse topologies, but provides a lower bound for comparison.

Figure 4 shows that for Overcast networks with greater than 200 nodes Overcast imposes somewhat less than twice as much network load as IP Multicast. In return for this extra load Overcast offers reliable delivery, immediate deployment, and future flexibility. For networks with few Overcast nodes, Overcast appears to impose a considerably higher network load than IP Multicast. This is a result of our optimistic lower bound on IP Multicast’s network load, which assumes that 50 randomly placed nodes in a 600 node network can be spanned by 49 links.

Another metric to measure the effectiveness of an application-level multicast technique is *stress*, proposed in [16]. Stress indicates the number of times that the same data traverses a particular physical link. By this metric, Overcast performs quite well with average stresses of between 1 and 1.2. We do not present detailed analysis of Overcast’s performance by this metric, however, because we believe

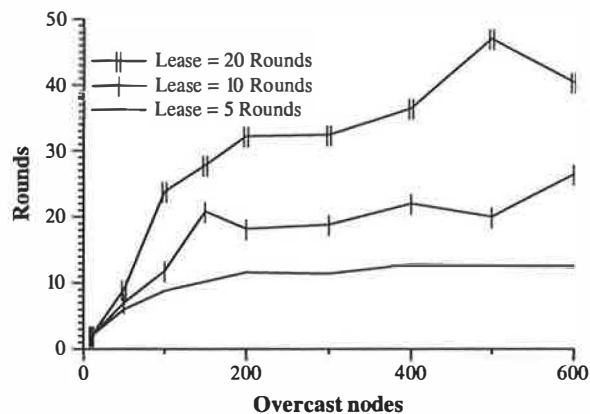


Figure 5: Number of rounds to reach a stable distribution tree as a function of the number of overcast nodes and the length of the lease period.

that network load is more telling for Overcast. That is, Overcast has quite low scores for average stress, but that metric does not describe how often a longer route was taken when a shorter route was available.

Another question is how fast the tree protocol converges to a stable distribution tree, assuming a stable underlying network. This is dependent on three parameters. The *round period* controls how long a node that has not yet determined a stable position in the hierarchy will wait before evaluating a new set of potential parents. The *reevaluation period* determines how long a node will wait before reevaluating its position in the hierarchy once it has obtained a stable position. Finally the *lease period* determines how long a parent will wait to hear from a child before reporting the child’s death.

For convenience, we measure all convergence times in terms of the fundamental unit, the round time. We also set the reevaluation period and lease period to the same value. Figure 5 shows how long Overcast requires to converge if an entire Overcast network is simultaneously activated. To demonstrate the effect of a changing reevaluation and lease period, we plot for the “standard” lease time—10 rounds, as well as longer and shorter periods. Lease periods shorter than five rounds are impractical because children actually renew their leases a small random number of rounds (between one and three) before their lease expires to avoid being thought dead. We expect that a round period on the order of 1-2 seconds will be practical for most applications.

We next measure convergence times for an existing Overcast network in which overcast nodes are added or fail. We simulate overcast networks of various

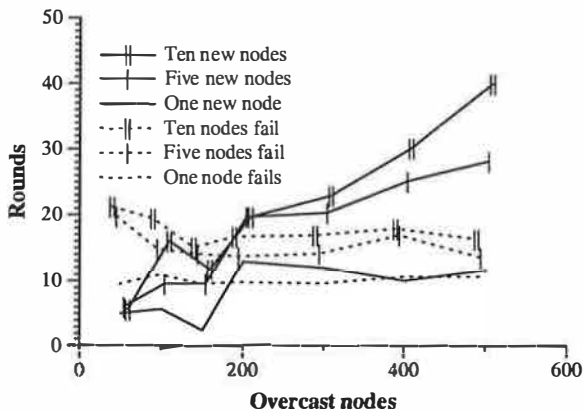


Figure 6: Number of rounds to recover a stable distribution tree as a function of the number of nodes that change state and the number of nodes in the network.

sizes until they quiesce, add and remove Overcast nodes, and then simulate the network until it quiesces once again. We measure the time, in rounds, for the network to quiesce after the changes. We measure for various numbers of additions and removals allowing us to assess the dependence of convergence on how many nodes have changed state. We measure only the backbone approach.

Figure 6 plots convergence times (using a 10 round lease time) against the number of overcast nodes in the network. The convergence time for node failures is quite modest. In all simulations the Overcast network reconverged after less than three lease times. Furthermore, the reconvergence time scaled well against both the number of nodes failing and the total number of nodes in the overcast network. In neither case was the convergence time even linearly affected.

For node additions, convergence times do appear more closely linked to the size of the Overcast network. This makes intuitive sense because new nodes are navigating the network to determine their best location. Even so, in all simulations fewer than five lease times are required. It is important to note that an Overcast network continues to function even while stabilizing. Performance may be somewhat impacted by increased measurement traffic and by TCP setup and tear down overhead as parents change, but such disruptions are localized.

5.2 Up/Down protocol

The goal of the up/down algorithm is to minimize the bandwidth required at the root node while maintaining timely status information for the entire network. Factors that affect the amount of bandwidth

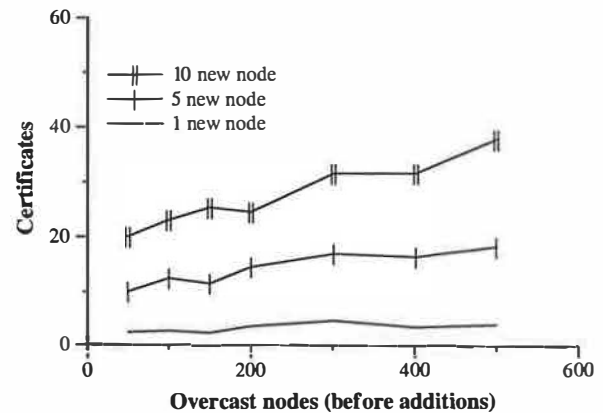


Figure 7: Certificates received at the root in response to node additions.

used include the size of the overcast network and the rate of topology changes. Topology changes occur when the properties of the underlying network change, nodes fail, or nodes are added. Therefore the up/down algorithm is evaluated by simulating overcast networks of various sizes in which various numbers of failures and additions occur.

To assess the up/down protocol's ability to provide timely status updates to the root without undue overhead we keep track of the number of certificates (for both "birth" and "death") that reach the root during the previous convergence tests. This is indicative of the bandwidth required at the root node to support an overcast network of the given size and is dependent on the amount of topology change induced by the additions and deletions.

Figure 7 graphs the number of certificates received by the root node in response to new nodes being brought up in the overcast network. Remember, the root may receive multiple certificates per node addition because the addition is likely to cause some topology reconfiguration. Each time a node picks a new parent that parent propagates a birth certificate. These results indicate that the number of certificates is quite modest: certainly no more than four certificates per node addition, usually approximately three. What is more important is that the number of certificates scales more closely to the number of new nodes than the size of the overcast network. This gives evidence that overcast can scale to large networks.

Similarly, Overcast requires few certificates to react to node failures. Figure 8 shows that in the common case, no more than four certificates are required per node failure. Again, because the number of certificates is proportional to the number of failures rather

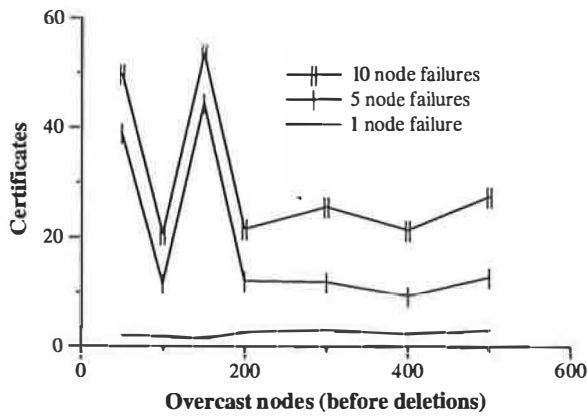


Figure 8: Certificates received at the root in response to node deletions.

than the size of the network, Overcast appears to offer the ability to scale to large networks.

On the other hand, Figure 8 shows that there are some cases that fall far outside the norm. The large spikes at 50 and 150 node networks with 5 and 10 failures occurred because of failures that happened to occur near the root. When a node with a substantial number of children chooses a new parent it must convey its entire set of descendants to its new parent. That parent then propagates the entire set. However, when the information reaches a node that already knows the relationships in question, the update is quashed. In these cases, because the reconfigurations occurred high in the tree there was no chance to quash the updates before they reached the root. In larger networks such failures are less likely.

6 Conclusions

We have described a simple tree-building protocol that yields bandwidth-efficient distribution trees for single-source multicast and our up/down protocol for providing timely status updates to the root of the distribution tree in scalable manner. Overcast implements these protocols in an overlay network over the existing Internet. The protocols allow Overcast networks to dynamically adapt to changes (such as congestion and failures) in the underlying network infrastructure and support large, reliable single-source multicast groups. Geographically-dispersed businesses have deployed Overcast nodes in small-scale Overcast networks for distribution of high-quality, on-demand video to unmodified desktops.

Simulation studies with topologies created with the Georgia Tech Internetwork Topology Models show

that Overcast networks work well on large-scale networks, supporting multicast groups of up to 12,000 members. Given these results and the low cost for Overcast nodes, we believe that putting computation and storage in the network fabric is a promising approach for adding new services to the Internet incrementally.

Acknowledgements

We thank Hari Balakrishnan for helpful input concerning the tree building algorithm; Suchitra Raman, Robert Morris, and our shepherd, Fred Douglis, for detailed comments that improved our presentation in many areas; and the many anonymous reviewers whose reviews helped us to see our work with fresh eyes.

References

- [1] FastForward Networks' broadcast overlay architecture. Technical report, FastForward, 2000. www.ffnet.com/pdfs/BOA-whitepaper6.PDF.
- [2] Elan Amir, Steven McCanne, and Randy H. Katz. An active service framework and its application to real time multimedia transcoding. In *Proc. ACM SIGCOMM Conference (SIGCOMM '98)*, pages 178–190, September 1998.
- [3] Yair Amir, Alec Peterson, and David Shaw. Seamlessly selecting the best copy from Internet-wide replicated web servers. In *The 12th International Symposium on Distributed Computing (DISC'98)*, pages 22–23, September 1998.
- [4] Michael Baentsch, Georg Molter, and Peter Sturm. Introducing application-level replication and naming into today's web. In *Proc. 5th International World Wide Web Conference*, May 1996.
- [5] A. Basso, C. Cranor, R. Gopalakrishnan, M. Green, C.R. Kalmanek, D. Shur, S. Sibal, C.J. Sreenan, and J.E. van der Merwe. PRISM, an IP-based architecture for broadband access to TV and other streaming media. In *Proc. IEEE International Workshop on Network and Operating System Support for Digital Audio and Video*, June 2000.
- [6] Azer Bestavros. Speculative data dissemination and service to reduce server load, network traffic, and response time in distributed information systems. In *Proc. of the 1996 International Conference on Data Engineering (ICDE '96)*, March 1996.
- [7] M. Blaze. *Caching in Large-Scale Distributed File Systems*. PhD thesis, Princeton University, January 1993.

- [8] Anawat Chankhunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. A hierarchical Internet object cache. In *Proc. USENIX 1996 Annual Technical Conference*, pages 153–164, January 1996.
- [9] Yatin Chawathe, Steven McCanne, and Eric Brewer. RMX: Reliable multicast for heterogeneous networks. In *Proc. IEEE Infocom*, March 2000.
- [10] P. Danzig, R. Hall, and M. Schwartz. A case for caching file objects inside internetworks. In *Proc. ACM SIGCOMM Conference (SIGCOMM '93)*, pages 239–248, September 1993.
- [11] S. E. Deering. *Multicast Routing in a Datagram Internetwork*. PhD thesis, Stanford University, December 1991.
- [12] R. Droms. Dynamic host configuration protocol. RFC 2131, Internet Engineering Task Force, March 1997. <ftp://ftp.ietf.org/rfc/rfc2131.txt>.
- [13] Paul Francis. Yoid: Your Own Internet Distribution. Technical report, ACIRI, April 2000. www.aciri.org/yoid.
- [14] J. Gwertzman and M. Seltzer. The case for geographical push-caching. In *Proc. 5th Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 51–57. IEEE Computer Society Technical Committee on Operating Systems, May 1995.
- [15] Hugh W. Holbrook and David R. Cheriton. IP multicast channels: EXPRESS support for large-scale single-source applications. In *Proc. ACM SIGCOMM Conference (SIGCOMM '99)*, pages 65–78, September 1999.
- [16] Yang hu Chu, Sanjay G. Rao, and Hui Zhang. A case for end system multicast. In *Proc. ACM SIGMETRICS Conference (SIGMETRICS '00)*, June 2000.
- [17] M. Frans Kaashoek, Robbert van Renesse, Hans van Staveren, and Andrew S. Tanenbaum. FLIP: an internetwork protocol for supporting distributed systems. *ACM Trans. Computer Systems*, 11(1):77–106, February 1993.
- [18] D. R. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc. 29th ACM Symposium on Theory of Computing*, pages 654–663, May 1997.
- [19] Steven McCanne and Van Jacobson. Receiver-driven layed multicast. In *Proc. ACM SIGCOMM Conference (SIGCOMM '96)*, pages 117–130, August 1996.
- [20] Jörg Nonnenmacher, Ernst W. Biersack, and Don Towsley. Parity-based loss recovery for reliable multicast transmission. In *Proc. ACM SIGCOMM Conference (SIGCOMM '97)*, pages 289–300, September 1997.
- [21] Stefan Savage, Tom Anderson, Amit Aggarwal, David Becker, Neal Cardwell, Andy Collins, Eric Hoffman, John Snell, Amin Vahdat, Geoff Voelker, and John Zahorjan. Detour: A case for informed Internet routing and transport. *IEEE Micro*, 19(1):50–59, January 1999.
- [22] Michael D. Schroeder, Andrew D. Birrell, Michael Burrows, Hal Murray, Roger M. Needham, Thomas L. Rodeheffer, Edwin H. Satterthwaite, and Charles P. Thacker. Autonet: A high-speed, self-configuring local area network using point-to-point links. *IEEE/ACM Trans. Networking*, 9(8):1318–1335, October 1991.
- [23] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.
- [24] J. Touch and S. Hotz. The X-bone (white paper). Technical report, SIS, May 1997. www.isi.edu/x-bone.
- [25] Ellen W. Zegura, Kenneth L. Calvert, and Samrat Bhattacharjee. How to model an internetwork. In *Proc. IEEE Infocom*, pages 40–52, March 1996.
- [26] Lixia Zhang, Scott Michel, Khoi Nguyen, Adam Rosenstein, Sally Floyd, and Van Jacobson. Adaptive web caching: Towards a new global caching architecture. In *Proc. 3rd International World Wide Web Caching Workshop*, June 1998.

System Support for Bandwidth Management and Content Adaptation in Internet Applications

David Andersen, Deepak Bansal, Dorothy Curtis, Srinivasan Seshan*, Hari Balakrishnan

M.I.T. Laboratory for Computer Science

Cambridge, MA 02139

{dga, bansal, dcurtis, srini, hari}@lcs.mit.edu

Abstract

This paper describes the implementation and evaluation of an operating system module, the Congestion Manager (CM), which provides integrated network flow management and exports a convenient programming interface that allows applications to be notified of, and adapt to, changing network conditions. We describe the API by which applications interface with the CM, and the architectural considerations that factored into the design. To evaluate the architecture and API, we describe our implementations of TCP; a streaming layered audio/video application; and an interactive audio application using the CM, and show that they achieve adaptive behavior without incurring much end-system overhead. All flows including TCP benefit from the sharing of congestion information, and applications are able to incorporate new functionality such as congestion control and adaptive behavior.

1 Introduction

The impressive scalability of the Internet infrastructure is in large part due to a design philosophy that advocates a simple architecture for the core of the network, with most of the intelligence and state management implemented in the end systems [10]. The service model provided by the network substrate is therefore primarily a “best-effort” one, which implies that packets may be lost, reordered or duplicated, and end-to-end delays may be variable. Congestion and accompanying packet loss are common in heterogeneous networks like the Internet because of overload, when demand for router resources, such as bandwidth and buffer space, exceeds what is available. Thus, end systems in the Internet should incorporate mechanisms for detecting and reacting to network congestion, probing for spare capacity when the network is uncongested, as well as managing their available bandwidth effectively.

Previous work has demonstrated that the result of uncontrolled congestion is a phenomenon commonly

called “congestion collapse” [8, 13]. Congestion collapse is largely alleviated today because the popular end-to-end Transmission Control Protocol (TCP) [30, 40] incorporates sound congestion avoidance and control algorithms. However, while TCP does implement congestion control [18], many applications including the Web [6, 12] use several logically different streams in parallel, resulting in multiple concurrent TCP connections between the same pair of hosts. As several researchers have shown [2, 3, 27, 28, 42], these concurrent connections compete with – rather than learn from – each other about network conditions to the same receiver, and end up being unfair to other applications that use fewer connections. The ability to share congestion information between concurrent flows is therefore a useful feature, one that promotes cooperation among different flows rather than adverse competition.

In today’s Internet is the increasing number of applications that do not use TCP as their underlying transport, because of the constraining reliability and ordering semantics imposed by its in-order byte-stream abstraction. Streaming audio and video [25, 34, 41] and customized image transport protocols are significant examples. Such applications use custom protocols that run over the User Datagram Protocol (UDP) [29], often without implementing any form of congestion control. The unchecked proliferation of such applications would have a significant adverse effect on the stability of the network [3, 8, 13].

Many Internet applications deliver documents and images or stream audio and video to end users and are *interactive* in nature. A simple but useful figure-of-merit for interactive content delivery is the end-to-end download latency; users typically wait no more than a few seconds before aborting a transfer if they do not observe progress. Therefore, it would be beneficial for content providers to adapt *what* they disseminate to the state of the network, so as not to exceed a threshold latency. Fortunately, such content adaptation is possible for most applications. Streaming audio and video applications typically encode information in a range of formats corresponding to different encoding (transmission) rates and degrees of loss resiliency. Image encoding formats accommodate a range of qualities to suit a

* Carnegie Mellon University, Pittsburgh, PA; srini@seshan.org

variety of client requirements.

Today, the implementor of an Internet content dissemination application has a challenging task: for her application to be safe for widespread Internet deployment, she must either use TCP and suffer the consequences of its fully-reliable, byte-stream abstraction, or use an application-specific protocol over UDP. With the latter option, she must re-implement congestion control mechanisms, thereby risking errors not just in the implementation of her protocol, but also in the implementation of the congestion controller. Furthermore, neither alternative allows for sharing congestion information across flows. Finally, the common application programming interface (API) classes for network applications—Berkeley sockets, streams, and Winsock [31]—do not expose any information about the state of the network to applications in a standard way¹. This makes it difficult for applications running on existing end host operating systems to make an informed decision, taking network variables into account, during content adaptation.

1.1 The Congestion Manager

Our previous work provided the rationale, initial design, and simulation of the Congestion Manager, an end-system architecture for sharing congestion information between multiple concurrent flows [3]. In this paper, we describe the implementation and evaluation of the CM in the Linux operating system. We focus on a version of the CM where the only changes made to the current IP stack are at the data sender, with feedback about congestion or successful data receptions being provided by the receiver CM applications to their sending peers, which communicate this information to the CM via an API. We present a summary of the API used by applications to adapt their transmissions to changing network conditions, and focus on those elements of the API that changed in the transition from the simulation to the implementation.

We evaluate the Congestion Manager by posing and answering several key questions:

Is its callback interface, used to inform applications of network state and other events, effective for a diverse set of applications to adapt without placing a significant burden on developers?

Because most robust congestion control algorithms rely on receiver feedback, it is natural to expect that a CM receiver is needed to inform the CM sender of successful transmissions and packet losses. However, to facilitate deployment, we have designed our system to take advantage of the fact that several protocols including TCP and other applications already incorporate some form of application-specific feedback, providing

¹ Utilities like `netstat` and `ifconfig` provide some information about devices, but not end-to-end performance information that can be used for adapting content.

the CM with the loss and timing information it needs to function effectively.

Using the CM API, we implement several case studies both in and out of the kernel, showing the applicability of the API to many different application architectures. Our implementation of a layered streaming audio/video application demonstrates that the CM architecture can be used to implement highly adaptive congestion controlled applications. Adaptation via the CM helps these applications achieve better performance and also be fair to other flows on the Internet.

We have also modified a legacy application—the Internet audio tool *vat* from the MASH toolkit [23]—to use the CM to perform adaptive real-time delivery. Since less than one hundred lines of source code modification was required to CM-enable this complex application and make it adapt to network conditions, we believe it demonstrates the ease with which the CM makes applications adaptive.

Is the congestion control correct?

As a trusted kernel module, the CM frees both transport protocols and applications from the burden of implementing congestion management. We show that the CM behaves in the same network-friendly manner as TCP for single flows. Furthermore, by integrating flow information between both kernel protocols and user applications, we ensure that an *ensemble* of concurrent flows is not an overly aggressive user of the network.

In today's off-the-shelf operating systems, does the CM place any performance limitations upon applications?

We find that our implementation of TCP (which uses the CM for its congestion control) has essentially the same performance as standard TCP, with the added benefits of integrated congestion management across flows, with only small (0-3%) CPU overhead.

In a CM system where no changes are made to the receiver protocol stack, UDP-based applications must implement a congestion feedback mechanism, resulting in more overhead compared to the TCP applications. However, we show that these applications remain viable, and that the architectural change and API calls reduce worst-case throughput by 0 - 25%, even for applications that desire fine-grained information about the network on a per-packet basis.

To our knowledge, this is the first implementation of a general application-independent system that combines integrated flow management with a convenient API to enable content adaptation. The end-result is that applications achieve the desirable congestion control properties of long-running TCP connections, together with the flexibility to adapt data transmissions to prevailing network conditions.

The rest of this paper is organized as follows. Section 2 describes our system architecture and implementation. Section 3 describes how network-adaptive applications can be engineered using the CM, while Section 4 presents the results of several experiments. In Section 5,

we discuss some miscellaneous details and open issues in the CM architecture. We survey related work in Section 6 and conclude with a summary in Section 7.

2 System Architecture and Implementation

The CM performs two important functions. First, it enables efficient multiplexing and congestion control by integrating congestion management across multiple flows. Second, it enables efficient application adaptation to congestion by exposing its knowledge of network conditions to applications. Most of the CM functionality in our Linux implementation is in-kernel; this choice makes it convenient to integrate congestion management across both TCP flows and other user-level protocols, since TCP is implemented in the kernel.

To perform efficient aggregation of congestion information across concurrent flows, the CM has to identify which flows potentially share a common bottleneck link *en route* to various receivers. In general, this is a difficult problem, since it requires an understanding of the paths taken by different flows. However, in today's Internet, all flows destined to the same end host take the same path in the common case, and we use this group of flows as the default granularity of flow aggregation². We call this group a *macroflow*: a group of flows that share the same congestion state, control algorithms, and state information in the CM. Each flow has a sending application that is responsible for its transmissions; we call this a *CM client*. CM clients are in-kernel protocols like TCP or user-space applications.

The CM incorporates a *congestion controller* that performs congestion avoidance and control on a per-macroflow basis. It uses a window-based algorithm that mimics TCP's additive-increase/multiplicative decrease (AIMD) scheme to ensure fairness to other TCP flows on the Internet. However, the modularity provided by the CM encourages experimentation with other non-AIMD schemes that may be better suited to specific data types such as audio or video.

While the congestion controller determines what the current window (rate) ought to be for each macroflow, a *scheduler* decides how this is apportioned among the constituent flows. Currently, our implementation uses a standard unweighted round-robin scheduler.

In-kernel CM clients such as a TCP sender use CM function calls to transmit data and learn about network conditions and events. In contrast, user-space clients interact with the CM using a portable, platform-independent API described in Section 2.1. A platform-dependent CM library, *libcm*, is responsible for interfacing between the kernel and these clients, and is described in Section 2.2. These components are shown in

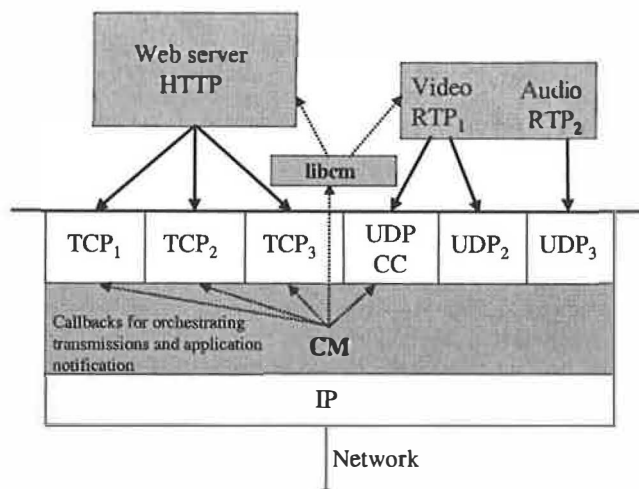


Figure 1. Architecture of the congestion manager at the data sender, showing the CM library and the CM. The dotted arrows show callbacks, and solid lines show the datapath. UDP-CC is a congestion-controlled UDP socket implemented using the CM.

Figure 1.

When a client opens a CM-enabled socket, the CM allocates a flow to it and assigns the flow to the appropriate macroflow based on its destination. The client initiates data transmission by requesting permission to send data. At some point in the future depending on the available rate, the CM issues a callback permitting the client to send data. The client then transmits data, and tells the CM it has done so. When the client receives feedback from the receiver about its past transmissions, it notifies the CM about these and continues.

When a client makes a request to send on a flow, the scheduler checks whether the corresponding macroflow's window is open. If so, the request is granted and the client notified, upon which it may send some data. Whenever any data is transmitted, the sender's IP layer notifies the CM, allowing it to "charge" the transmission to the appropriate macroflow. When the client receives feedback from its remote counterpart, it informs the CM of the loss rate, number of bytes transmitted correctly, and the observed round trip time. On a successful transmission, the CM opens up the window according to its congestion management algorithm and grants the next, if any, pending request on a flow associated with this macroflow. The scheduler also has a timer-driven component to perform background tasks and error handling.

² This is not strictly true in the presence of network-layer differentiated services. We address this issue later in this section and in Section 5.

2.1 CM API

The CM API is specified as a set of functions and callbacks that a client uses to interface with the CM. It specifies functions for managing state, for performing data transmissions, for applications to inform the CM of losses, for querying the CM about network state, and for constructing and splitting macroflows if the default per-destination aggregation is unsuitable for an application. The CM API is discussed in detail in [3], which presents the design rationale for the Congestion Manager. Here we provide an overview of the API and a discussion of those features which changed during the transition from simulation to implementation.

2.1.1 State management

All CM applications call `cm_open()` before using the CM, passing the source and destination addresses and transport-layer port numbers, in the form of a `struct sockaddr`. The original CM API required only a destination address, but the source address specification was necessary to handle multihomed hosts. `cm_open` returns a flow identifier (`cm_flowid`), which is used as a handle for all future CM calls. Applications may call `cm_mtu(cm_flowid)` to obtain the maximum transmission unit to a destination. When a flow terminates, the application should call `cm_close(cm_flowid)`.

2.1.2 Data transmission

There are three ways in which an application can use the CM to transmit data. These allow a variety of adaptation strategies, depending on the nature of the client application and its software structure.

- (i) **Buffered send.** This API uses a conventional `write()` or `sendto()` call, but the resulting data transmission is paced by the Congestion Manager. We use this to implement a generic congestion-controlled UDP socket (without content adaptation), useful for bulk transmissions that do not require TCP-style reliability or fine-grained control over what data gets sent at a given point in time.
- (ii) **Request/callback.** This is the preferred mode of communication for adaptive senders that are based on the ALF (Application-Level Framing [11]) principle. Here, the client does not send data via the CM; rather, it calls `cm_request(cm_flowid)` and expects a notification via the `cmapp_send(cm_flowid)` callback when this request is granted by the CM, at which time the client transmits its data. This approach puts the sender in firm control of deciding what to transmit at a given time, and allows the sender to adapt to sudden changes in network performance, which is hard to do in a conventional buffered transmission API. The client callback is a grant

for the flow to send up to MTU bytes of data. Each call to `cm_request()` is an implicit request for sending up to MTU bytes, which simplifies the internal implementation of the CM. This API is ideally suited for an implementation of TCP, since it needs to make a decision at each stage about whether to retransmit a segment or send a new one. In the implementation, the `cmapp_send` callback now provides the client with the ID of the flow that may transmit. To allow for client programming flexibility, the client may now specify its callback function via `cm_register_send()`.

- (iii) **Rate callback.** A self-timed application transmitting on a fixed schedule may receive callbacks from the CM notifying it when the parameters of its communication channel have changed, so that it can change the frequency of its timer loop or its packet size. The CM informs the client of the rate, round-trip time, and packet loss rate for a flow via the `cmapp_update()` callback. During implementation, we added a registration function, `cm_register_update()` to select the rate callback function, and the `cm_thresh(down,up)` function: If the rate reduces by a factor of `down` or increases by a factor of `up`, the CM calls `cmapp_update()`. This transmission API is ideally suited for streaming layered audio and video applications.

2.1.3 Application notifications

One of the goals of our work was to investigate a CM implementation that requires no changes at the receiver. Performing congestion management requires feedback about transmissions: TCP provides this feedback automatically; some UDP applications may need to be modified to do so, but without any system-wide changes. Senders must then inform the CM about the number of sent and received packets, type of congestion loss if any, and a round-trip time sample using the `cm_update(cm_flowid, nsent, nrcd, lossmode, rtt)` function. The CM distinguishes between “persistent” congestion as would occur on a TCP timeout, versus “transient” congestion when only one packet in a window is lost. It also allows congestion to be notified using Explicit Congestion Notification (ECN) [32], which uses packet markings rather than drops to infer congestion.

To perform accurate bookkeeping of the congestion window and outstanding bytes for a macroflow, the CM needs to know of each successful transmission from the host. Rather than encumber clients with reporting this information, we modify the IP output routine to call `cm_notify(cm_flowid, nsent)` on each transmission. (The IP layer obtains the `cm_flowid` using a well-defined CM interface that takes the flow parameters (addresses, ports, protocol field) as arguments.) However, if a client decides not to transmit any data upon a `cmapp_send()` callback invocation, it is expected to call

`cm_notify(dst, 0)` to allow the CM to permit some other flows on the macroflow to transmit data.

2.1.4 Querying

If a client wishes to learn about its (per-flow) available bandwidth and round-trip time, it can use the `cm_query()` call, which returns these quantities. This is especially useful at the beginning of a stream when clients can make an informed decision about the data encoding to transmit (e.g., a large color or smaller grey-scale image).

2.2 libcm: The CM library

The CM library provides users with the convenience of a callback-based API while separating them from the details of how the kernel to user callbacks are implemented. While direct function callbacks are convenient and efficient in the same address space, as is the case when the kernel TCP is a client of the CM, callbacks from the kernel to user code in conventional operating systems are more difficult. A key decision in the implementation of `libcm` was choosing a kernel/user interface that maximizes portability, and minimizes both performance overhead and the difficulty of integration with existing applications. The resulting *internal* interface between `libcm` and the kernel is:

1. `select()` on a single per-application CM control socket. The write bit indicates that a flow may send data, and the exception bit indicates that network conditions have changed.
2. Perform an `ioctl` to extract a list of all flow IDs which may send, or to receive the current network conditions for a flow.

Note that client programs of the CM do not see this interface; they see only the standard `cm_*` functions provided by `libcm`. The use of sockets or signals does change the way the application's event handling loop interacts with `libcm`; after passing the socket into `libcm`, the library performs the appropriate `ioctl`s and then calls back into the application.

2.2.1 Implementation alternatives

We considered a number of mechanisms with which to implement `libcm`. In this section, we discuss our reasons for choosing the control-socket+select+ioctl approach.

While much research has focused on reducing the cost of crossing the user/kernel boundary (extensible kernels in SPIN [7], fast, generic IPC in Mach [5], etc.) many conventional operating systems remain limited to more primitive methods for kernel-to-user notification, each with their own advantages and disadvantages. While functionality like the Mach port set-based

IPC would be ideal for our purposes, pragmatically we considered four common mechanisms for kernel to user communication: Signals, system calls, semaphores, and sockets. A discussion of the merits of each follows.

Signals have several immediate drawbacks. First, if the CM were to appropriate an existing signal for its own use, it might conflict with an application using the same signal. Avoiding this conflict would require the standardization of a new signal type, a process both slow and of questionable value, given the existence of better alternatives. Second, the cost to an application to receive a signal is relatively high, and some legacy applications may not be signal-safe. While the new POSIX 1003.1b [17] soft realtime signals allow delivering a 32-bit quantity with a signal, applications would need to follow up a signal with a system call to obtain all of the information the kernel wished to deliver, since multiple flows may become ready at once. For these reasons, we consider mandating the use of signals the wrong course for implementing the kernel to user callbacks. However, we provide an *option* for processes to receive a SIGIO when their control socket status changes, akin to POSIX asynchronous I/O.

System calls that block do not integrate well with applications that already have their own event loop, since without polling, applications cannot wait on the results of multiple system calls. A system call is able to return immediately with the data the user needs, but the impediments it poses to application integration are large. System calls would work well in a threaded environment, but this presupposes threading support, and the select-based mechanism we describe below can be used in a threaded system without major additional overhead.

Semaphores suffer from the immediate drawback that they are not commonly used in network applications. For an application that uses `semop` on an array of semaphores as its event loop, a CM semaphore might be the best implementation avenue, for many of the same reasons that we chose sockets for network-adaptive applications. However, most network applications use socket sets instead of semaphore sets, and sockets have a few other benefits, which we discuss next.

Sockets provide a well-defined and flexible interface for applications in the form of the `select()` system call, though they have a downside similar to that of signals: an application wishing to receive a notification via a socket in a non-blocking manner must `select()` on the socket, and then perform a system call to obtain data from the socket. However, a select-based interface meshes well with many network applications that already have a select-loop based architecture. Utilizing a control socket also helps restrict the code changes caused by the CM to the networking stack.

Finally, we decided to use a single control socket instead of one control socket per flow to avoid unnecessary overhead in applications with large numbers of open socket descriptors, such as `select()`-based web-

servers and caches. Because some aspects of select scale linearly with the number of descriptors, and many operating systems have limits on the number of open descriptors, we deemed doubling the socket load for high-performance network applications a bad idea.

2.2.2 *Extracting data from the socket*

Select provides notification that “some event” has occurred. In theory, 7 different events could be sent by abusing the read, write, and exception bits, but applications need to extract more information than this. The CM provides two types of callbacks. Generally speaking, the first is a “permission to send” callback for a particular flow. To maintain even distribution of bandwidth between flows, a loose ordering should be preserved with these messages, but exact ordering is unimportant provided no flows are ignored until the application receives further updates (thereby starving the flows). If multiple permission notifications occur, the application should receive *all* of them so it can send data on all available flows. The second callback is a “status changed” notification. If multiple status changes occur before the application obtains this data from the kernel, then only the *current* status matters.

The weak ordering and lack of history prompted us to choose an `ioctl`-based query instead of a read or message queue interface, minimizing the state that must be maintained in the kernel. Status updates simply return the current CM-maintained network state estimate, and “who can send” queries perform a select-like operation on the flows maintained by the kernel, requiring no extra state, instead of a potentially expensive per-process message queue or data stream. Returning all available flows has an added benefit of reducing the number of system calls that must be made if several flows become ready simultaneously.

3 Engineering Network-adaptive Applications

In this section, we describe several different classes of applications, and describe the ways those applications can make use of the CM. We explore two in-kernel clients, and several user-space data server programs, and examine the task of integrating each with the CM.

3.1 Software Architecture Issues

Typical network applications fall into one of several categories:

- Data-driven: Applications that transmit prespecified data, such as a single file, then exit.
- Synchronous event-driven: Self-timed data delivery servers, like streaming audio servers.

- Asynchronous event-driven: File servers (http, ftp) and other network-clocked applications.

The CM library provides several options for adaptive applications that wish to make use of its services:

1. Data-driven applications may use the buffered API to efficiently pace their data transmissions.
2. An application may operate in an entirely callback-based manner by allowing `libcm` to provide its own event loop, calling into the application when flows are ready. This is most useful for applications coded with the CM in mind.
3. Signal-driven applications may request a SIGIO notification from the CM when an event occurs.
4. Applications with select-based event loops can simply add the CM control socket into their select set, and call the `libcm` dispatcher when the socket is ready. Rate-clocked applications (or polling-based applications) can perform a similar non-blocking select test on the descriptor when they awaken to send data, or, if they sleep, can replace the sleep with a timed blocking select call.
5. Applications may poll the CM on their own schedule.

The remainder of this section describes how particular clients use different CM APIs, from the low-bandwidth vat audio application, to the performance-critical kernel TCP implementation. Note that all UDP-based clients must implement application level data acknowledgements in order to make use of the CM.

3.2 TCP

We implemented TCP as an in-kernel CM client. TCP/CM offloads all congestion control to the CM, while retaining all other TCP functionality (connection establishment and termination, loss recovery and protocol state handling). TCP uses the request/callback API as low-overhead direct function calls in the same protection domain. This gives TCP the tight control it needs over packet scheduling. For example, while the arrival of a new acknowledgement typically causes TCP to transmit new data, the arrival of three duplicate ACKs causes TCP to retransmit an old packet.

Connection creation. When TCP creates a new connection via either `accept` (inbound) or `connect` (outbound), it calls `cm_open()` to associate the TCP connection with a CM flow. Thereafter, the pacing of outgoing data on this connection is controlled by the CM. When application data becomes available, after performing all the non-congestion-related checks (e.g., the Nagle algorithm [40], etc.) data is queued and `cm_request()` is called for the flow. When the CM scheduler schedules the flow for transmission,

the `cmapp_send()` routine for TCP is called. The `cmapp_send()` for TCP transmits any retransmission from the retransmission queue. Otherwise, it transmits the data present in the transmit socket buffer by sending up to one maximum segment size of data per call. Finally, the IP output routine calls `cm_notify()` when the data is actually sent out.

TCP input. The TCP input routines now feed-back to the CM. Round trip time (RTT) sample collection is done as usual using either RFC 1323 timestamps [19] or Karn's algorithm [21] and is passed to CM via `cm_update()`. The smoothed estimates of the RTT (`srtt`) and round-trip time deviation are calculated by the CM, which can now obtain a better average by combining samples from different connections to the same receiver. This is available to each TCP connection via `cm_query()`, and is useful in loss recovery.

Data acknowledgements. On arrival of an ACK for new data, the TCP sender calls `cm_update()` to inform the CM of a successful transmission. Duplicate acknowledgements cause TCP to check its dupack count (`dup_acks`). If `dup_acks < 3`, then TCP does nothing. If `dup_acks == 3`, then TCP assumes a simple, congestion-caused packet loss, and calls `cm_update` to inform the CM. TCP also enqueues a retransmission of the lost segment and calls `cm_request()`. If `dup_acks > 3`, TCP assumes that a segment reached the receiver and caused this ACK to be sent. It therefore calls `cm_update()`. Unlike duplicate ACKs, the expiration of the TCP retransmission timer notifies the sender of a more serious batch of losses, so it calls `cm_update` with the `CMLOSTFEEDBACK` option set to signify the occurrence of persistent congestion to the CM. TCP also enqueues a retransmission of the lost segment and calls `cm_request()`.

TCP/CM Implementation. The integration of TCP and the CM required less than 100 lines of changes to the existing TCP code, demonstrating both the flexibility of the CM API and the low programmer overhead of implementing a complex protocol with the Congestion Manager.

3.3 Congestion-controlled UDP sockets

The CM also provides congestion-controlled UDP sockets. They provide the same functionality as standard Berkeley UDP sockets, but instead of immediately sending the data from the kernel packet queue to lower layers for transmission, the buffered socket implementation schedules its packet output via CM callbacks. When a CM UDP socket is created, it is bound to a particular flow. When data enters the packet queue, the kernel calls `cm_request()` on the flow associated with the socket. When the CM schedules this flow for transmission, it calls `udp_ccappsend()` in the CM UDP module. This function transmits one MTU from the packet queue, and requests another callback if packets remain. The in-kernel implementation of the CM UDP API adds

no data copies or queue structures, and supports all standard UDP options. Modifying existing applications to use this API requires only providing feedback to the CM, and setting a socket option on the socket.

A typical client of the CM UDP sockets will behave as follows, after its usual network socket initialization:

```
flow = cm_open(dst, port)
setsockopt(flow, ..., CM_BUF)
loop:
    <send data on flow>
    <receive data acknowledgements>
    cm_update(flow, sent, received, ...)
```

3.4 Streaming Layered Audio and Video

Streaming layered audio or video applications that have a number of discrete rates at which they can transmit data are well-served by the CM rate callbacks. Instead of requiring a comparatively expensive notification for each transmission, these applications are instead notified only in the rare event that their network conditions change significantly. Layered applications open their usual UDP socket, and call `cm_open()` to obtain a control socket. They operate in their own clocked event loop while listening for status changes on either their control socket or via a SIGIO signal. They use `cm_thresh()` to inform the CM about network changes for which they should receive callbacks.

3.5 Real-time Adaptive Applications

Applications that desire last-minute control over their data transmission (i.e. those that do not want any buffering inside the kernel) use the request callback API provided by the CM. When given permission to transmit via the `cmapp_send()` callback from the CM, they may use `cm_query()` to discover the current network conditions and adapt their content based on that. Other servers may simply wish to send the most up-to-date content possible, and so will defer their data collection until they know they can send it. The rough sequence of CM calls that are made to achieve this in the application are:

```
flow = cm_open(dst)
cm_request(flow)
<receive cmapp_send() callback from libcm>
cm_query(flow, ...)
<send data>
<receive data acks>
cm_update(flow, sent, lost, ...)
```

Other options exist for applications that wish to exploit the unique nature of their network utilization to reduce the overhead of using the services of the Congestion Manager. We discuss one such option below in the manner in which we adapted the vat interactive audio application to use the CM.

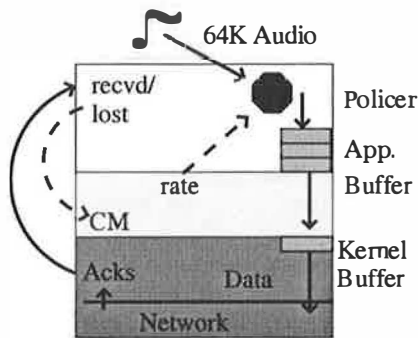


Figure 2. The adaptive vat architecture

3.6 Interactive Real-time Audio

The vat application provides a constant bit-rate source of interactive audio. Its inability to downsample its audio reduces the avenues it has available for bandwidth adaptation. Therefore, the best way to make vat behave in a network-friendly and backwards compatible manner is to preemptively drop packets to match the available network bandwidth. There are, of course, complications. Network applications experience two types of variation in available network bandwidth: long term variations due to changes in actual bandwidth, and short term variations due to the probing mechanisms of the congestion control algorithm. Short-term variation is typically dealt with by buffering. Unfortunately, buffering, especially FIFO buffering with drop-tail behavior, the de-facto standard for kernel buffers and network router buffers, can result in long delay and significant delay variation, both of which are detrimental to vat's audio quality. Vat, therefore, needs to act like an ALF application, managing its own buffer space with drop-from-head behavior when the queue is full.

The resulting architecture is detailed in figure 2. The input audio stream is first sent to a policer, which provides long-term adaptation via preemptive packet dropping. The policer outputs into the application level buffer, which can be configured in various sizes and drop policies. This buffer feeds into the kernel buffer on-demand as packets are available for transmission.

4 Evaluation

This section describes several experiments that quantify the costs and benefits of our CM implementation. Our experiments show that using the Congestion Manager in the kernel has minimal costs, and that even the worst-case overhead of the request/callback user-space API is acceptably small.

The tests were performed on the Utah Network Testbed [22] using 600MHz Intel Pentium III processors, 128MB PC100 ECC SDRAM, and Intel EtherExpress Pro/100B Ethernet cards, connected via 100Mbps

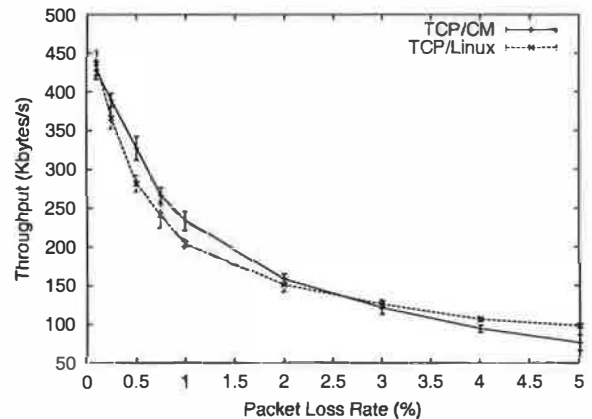


Figure 3. Comparing throughput vs. loss for TCP/CM and TCP/Linux. Rates are for a 10Mbps link with a 60ms RTT.

Ethernet through an Intel Express 510T switch, with Dummynet channel simulation. CM tests were run on Linux 2.2.9, with Linux and FreeBSD clients.

To ensure the proper behavior of a flow, the congestion control algorithm must behave in a "TCP-compatible" [8] manner. The CM implements a TCP-style window-based AIMD algorithm with slow start. It shares bandwidth between eligible flows in a round-robin manner with equal weights on the flows.

Figure 3 shows the throughput achieved by the Linux TCP implementation (TCP/Linux) and TCP with congestion control performed by the CM (TCP/CM). The linux kernel against which we compare has two algorithmic differences from the Congestion Manager: It starts its initial window at 2 packets, and it assumes that each ACK is for a full MTU. The Congestion Manager instead performs byte-counting for its AIMD algorithm. The first issue is Linux-specific, and the last is a feature of the CM.

4.1 Kernel Overhead

To measure the kernel overhead, we measured the CPU and throughput differences between the optimized TCP/Linux and TCP/CM. The midrange machines used in our test environment are sufficiently powerful to saturate a 100Mbps Ethernet with TCP traffic.

There are two components to the overhead imposed by the congestion manager: The cost of performing accounting as data is exchanged on a connection, and a one-time connection setup cost for creating CM data structures. A microbenchmark of the connection establishment time of a TCP/CM vs. TCP/Linux indicates that there is no appreciable difference in connection setup times.

We used long (megabytes to gigabytes) connections with the `ttcp` utility to determine the long-term costs

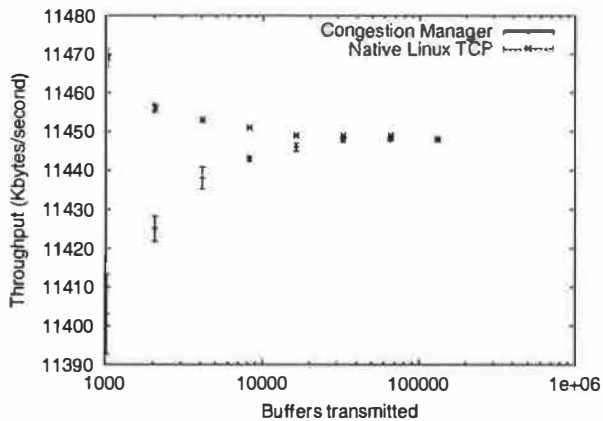


Figure 4. 100Mbps TCP throughput comparison. Note that the absolute difference in the worst case between the Congestion Manager and the native TCP is only 0.5% and that the Y axis begins at 11 megabytes per second.

imposed by the congestion manager. The impact of the CM on extremely long term throughput was negligible: in a 1 gigabyte transfer, the congestion manager achieved identical performance (91.6 Mbps) as native Linux. On shorter runs, the throughput of the CM diverged slightly from that of Linux, but only by 0.5%. The throughput rates are shown in figure 4. The difference is due to the CM using an initial window of 1 MTU and Linux using 2 MTU, not CPU overhead.

Because both implementations are able to saturate the network connection, we looked at the CPU utilization during these transmissions to determine the steady-state overhead imposed by the Congestion Manager. In figure 5 we see that the CPU difference between TCP/Linux and TCP/CM converges to slightly less than 1%.

4.2 User-space API Overhead

The overhead incurred by our adaptation API occurs primarily because the applications must process their ACKs in user-space instead of in the kernel. Therefore, these programs incur extra data copies and user/kernel boundary crossings. To quantify this overhead, our test programs sent packets of specified sizes on a UDP socket, and waited for acknowledgement packets from the server. We compare these programs to a webserver-like TCP client which send data to the server, and performed a `select()` on its socket to determine if the server has sent any data back. To facilitate comparison, we disabled delayed ACKs for the one TCP test to ensure that our packet counts were identical.

Figure 6 shows the wall-clock time required to send and process the acknowledgement for a packet, based on transmitting 200,000 packets. For comparison, we in-

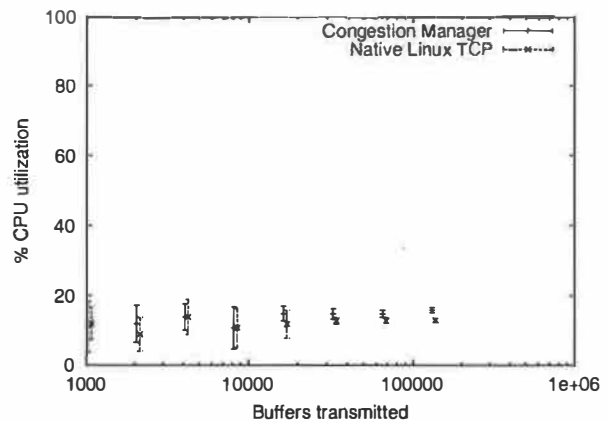


Figure 5. CPU overhead comparison between TCP/Linux and TCP/CM. For long connections, the CPU overhead converges to slightly under 1% for the unoptimized implementation of the CM.

ALF/noconnect	1 <code>cm_notify</code> (ioctl)
ALF	1 <code>cm_request</code> (ioctl) 1 extra socket
Buffered	1 <code>recv</code> , 2 <code>gettimeofday</code>
TCP/CM	--baseline--

Table 1. Cumulative sources of overhead for different APIs using the Congestion Manager relative to sending data with TCP.

clude TCP statistics as well, where the TCP programs set the maximum segment size to achieve identical network performance. The “nodelay” variant is TCP without delayed acks. The tests were run on a 100Mbps network on which no losses occurred.

Table 1 breaks down the sources of overhead for using the different APIs. Using the CM with UDP requires that applications compute the round-trip-time (RTT) of their packets, requiring a system call to `gettimeofday`, and requires that they process their ACKs in user-space, requiring a system call to `recv` and the accompanying data copy into their address space. The ALF API further requires that the application obtain an additional control socket and select upon it, and that it make an explicit call to `cm_request` before transmitting data. Finally, if the kernel is unable to determine the flow to which to charge the transmission, as with an unconnected UDP socket, the application must explicitly call `cm_notify`.

These test cases represent the worst-case behavior of serving a single high-bandwidth client, because no aggregation of requests to the CM may occur. The CM programs can achieve similar reductions in processing time by using delayed acks, so the real API overhead can be determined by comparing the ALF/noconnect case to the TCP/CM case. For 168 byte packets,

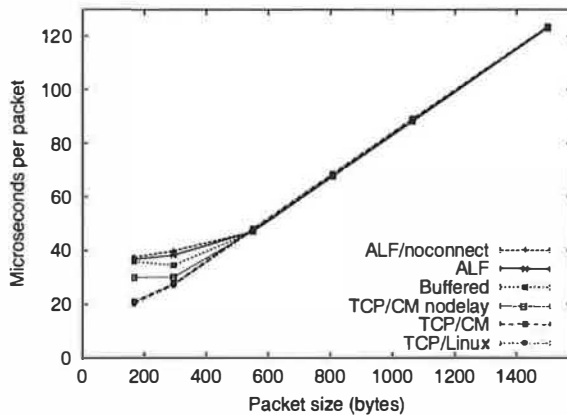


Figure 6. API throughput comparison on a 100Mbps link. The worst-case throughput reduction incurred by the CM is 25% from TCP/CM nodelay to ALF/noconnect.

ALF/noconnect results in a 25% reduction in throughput relative to TCP without delayed ACKs.

4.3 Benefits of Sharing

One benefit of integrating congestion information with the CM is immediately clear. A client that sequentially fetches files from a webserver with a new TCP connection each time loses its prior congestion information, but with concurrent connections with the CM, the server is able to use this information to start subsequent connections with more accurate congestion windows. Figure 7 shows a test we performed across the vBNS between MIT and the University of Utah, where an unmodified (non-CM) client performed 9 retrievals of the same 128k file with a 500ms delay between retrievals, resulting in a 40% improvement in the transfer time for the later requests. (Other file sizes and delays yield similar results, so long as they overlap. The benefits are comparatively greater for smaller files). The CM requires an additional RTT (75ms) for the first transfer, because Linux sets its initial congestion window to 2 MTUs instead of 1. This pattern of multiple connections is still quite common in webserver despite the adoption of persistent connections: Many browsers open 4 concurrent connections to a server, and many client/server combinations do not support persistent connections. Persistent connections [28] provide similar performance benefits, but suffer from their own drawbacks, which we discuss in section 6.

4.4 Adaptive Applications

In this section, we demonstrate some of the network adaptive behaviors enabled by the CM.

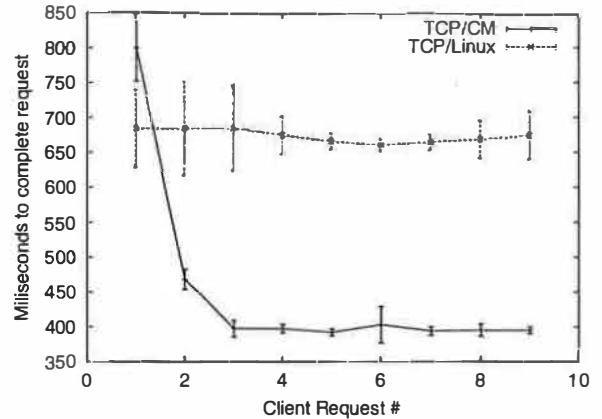


Figure 7. Sharing TCP state: The client requests the same file 9 times with a 500ms delay between request initiations. By sharing congestion information and avoiding slow-start, the CM-enabled server is able to provide faster service for subsequent requests, despite a smaller initial congestion window.

As noted earlier, applications that require tight control over data scheduling use the request/callback (ALF) API, and are notified by the CM as soon as they can transmit data. The behavior of an adaptive layering application run across the vBNS using this API is shown in figure 8. This application chooses a layer to transmit based upon the current rate, but sends packets as rapidly as possible to allow its client to buffer more data. We see that the CM is able to provide sufficient information to the application to allow it to adapt properly to the network conditions.

For self-clocked applications that base their transmitted data upon the bandwidth to the client (such as conventional layered audio servers), the CM rate callback mechanism provides a low-overhead mechanism for adaptation, and allows clients to specify thresholds for the notification callbacks. Figure 9 shows application adaptation using rate callbacks for a connection between MIT and the University of Utah. Here, the application decides which of the four layers it should send based on notifications from the CM about rate changes.

From figures 8 and 9, we see from the increased oscillation rate in the transmitted layer that the ALF application is more responsive to smaller changes in available bandwidth, whereas the rate callback application relies occasionally on short-term kernel buffering for smoothing. There is an overhead vs. functionality trade-off in the decision of which API to use, given the higher overhead of the ALF API, but applications face a more important decision about the behavior they desire.

Some applications may be concerned about the overhead from receiver feedback. To mitigate this, an application may delay sending feedback; we see this in a minor and inflexible way with TCP delayed acks. In

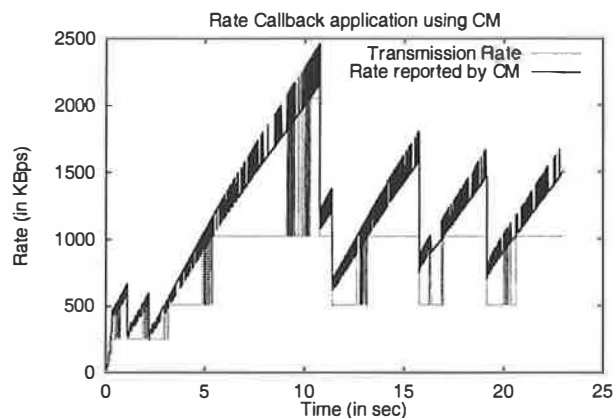


Figure 8. Bandwidth perceived by an adaptive layered application using the request callback (ALF) API.

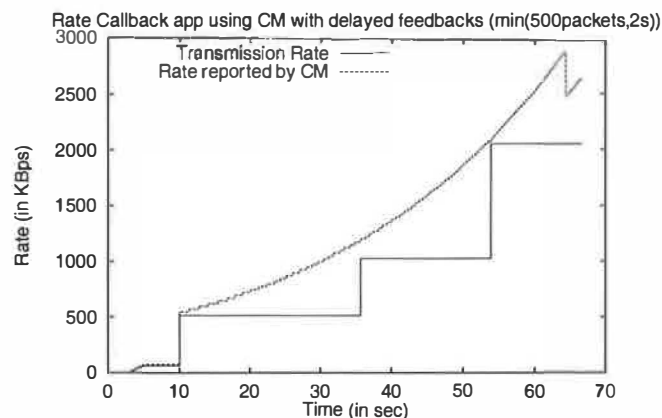


Figure 10. Adaptive layered application using rate callback API with delayed feedback

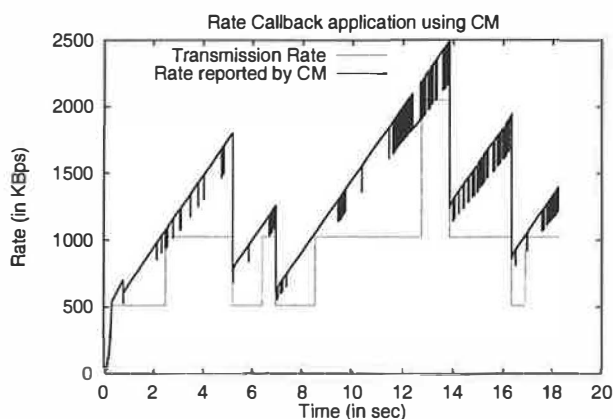


Figure 9. Bandwidth perceived by an adaptive layered application using the rate callback API.

figure 10, we see that delaying feedback to the CM causes burstiness in the reported bandwidth. Here, the feedback by the receiver was delayed by $\min(500 \text{ acks}, 2000\text{ms})$. The initial slow start is delayed by 2s waiting for the application, then the update causes a large rate change. Once the pipe is sufficiently full, 500 acks come relatively rapidly, and the normal, though bursty, non-timeout behavior resumes.

5 Discussion

We have shown several benefits of integrated flow management and the adaptation API, and have explored the design features that make the API easy to use. This section describes an optimization useful for busy servers, and discusses some drawbacks and limitations of the current CM architecture.

Optimizations. Servers with large numbers of concurrent clients are often very sensitive to the overhead caused by multiple kernel boundary crossings. To reduce this overhead, we can batch several sockets into the same `cm_request` call with the `cm_bulk_request` call, and likewise for `query`, `notify`, and `update` calls.

By multiplexing control information for many sockets on each CM call, the overhead from kernel crossings is mitigated at the expense of managing more complicated data structures for the CM interface. Bulk querying is already performed in `libcm` when multiple flows are ready during a single `ioctl` to determine which flows can send data, but this completes the interface.

Trust issues. Because our goal was an architecture that did not require modifications to receivers, we devised a system where applications provide feedback using the `cm_update()` call. The consequence of this is that there is a potential for misuse, due to bugs or malice. For example, the CM client could repeatedly misinform the CM about the absence of congestion along a path and obtain higher bandwidth. This does not increase the vulnerability of the Internet to such problems, because such abuse is already trivial. More important are situations where users on the same machine could potentially interfere with each other. To prevent this, the Congestion Manager would need to ensure that only kernel-mediated (e.g. TCP) flows belonging to different users can belong in the same macroflow. Our current implementation does not make an attempt to provide this protection. Savage [37] presents several methods by which a malicious receiver can defeat congestion control. The solutions he proposes can be easily used with the CM; we have already implemented byte-counting to prevent ACK division.

Macroflow construction. When differentiated services, or any system which provides different service to flows between the same pair of hosts, start being deployed, the CM would have to reconsider the default

choice of a macroflow. We expect to be able to gain some benefit by including the IP differentiated-services field in deciding the composition of a macroflow.

Finally, we observe that remote LANs are not often the bottleneck for an outside communicator. As suggested in [42, 36] among others, aggregating congestion information about remote sites with a shared bottleneck and sharing this information with local peers may benefit both users and the network itself. A macroflow may thus be extended to cover multiple destination hosts behind the same shared bottleneck link. Efficiently determining such bottlenecks remains an open research problem.

Limitations. The current CM architecture is designed only to handle unicast flows. The problem of congestion control for multicast flows is a much more difficult problem which we deliberately avoid. UDP applications using the CM are required to perform their own loss detection, requiring potential additional application complexity. Implementing the Congestion Manager protocol discussed in [3] would eliminate this need, but remains to be studied.

6 Related work

Designing adaptive network applications has been an active area of research for the past several years. In 1990, Clark and Tennenhouse [11] advocated the use of *application-level framing* (ALF) for designing network protocols, where protocol data units are chosen in concert with the application. Using this approach, an application can have a greater influence over deciding how loss recovery occurs than in the traditional layered approach. The ALF philosophy has been used with great benefit in the design of several multicast transport protocols including the Real-time Transport Protocol (RTP) [38], frameworks for reliable multicast [14, 33], and Internet video [24, 35].

Adaptation APIs in the context of mobile information access were explored in the Odyssey system [26]. Implemented as a user-level module in the NetBSD operating system, Odyssey provides API calls by which applications can manage system resources, with upcalls to applications informing them when changes occur in the resources that are available. In contrast, our CM system is implemented in-kernel since it has to manage and share resources across applications (e.g., TCP) that are already in-kernel. This necessitates a different approach to handling application callbacks. In addition, the CM approach to measuring bandwidth and other network conditions is tied to the congestion avoidance and control algorithms, as compared to the instrumentation of the user-level RPC mechanism in Odyssey. We believe that our approach to providing adaptation information for bandwidth, round-trip time, and loss rate complements Odyssey's management of disk space, CPU, and battery power.

The CM system uses application callbacks or *upcalls* as an abstraction, an old idea in operating systems. Clark describes upcalls in the Swift operating system, where the motivation is a lower layer of a protocol stack synchronously invoking a higher-layer function across a protection boundary [9]. The Mach system used the notion of *ports*, a generic communication abstraction for fast inter-process communication (IPC). POSIX specifies a standard way of passing "soft real-time signals" that can be used to send a notification to a user-level process, but it restricts the amount of data that can be communicated to a 32-bit quantity.

Event delivery abstractions for mobile computing have been explored in [1], where "monitored" events are tracked using polling and "triggered" events (e.g., PC card insertion) are notified using IPC. This work defines a language-level mechanism based on C++ objects for event registration, delivery, and handling. This system is implemented in Mach using ports for IPC.

Our approach is to use a `select()` call on a control socket to communicate information between kernel and user-level. The recent work of Banga *et al.* [4] to improve the performance of this type of event delivery can be used to further improve our performance.

The Microsoft Winsock implementation is largely callback-based, but here callbacks are implemented as conventional function calls since Winsock is a user-level library within the same protection boundary as the application [31]. The main reason we did not implement the CM as a user-level daemon was because TCP is already implemented in-kernel in most UNIX operating systems, and it is important to share network information across TCP flows.

Quality-of-service (QoS) interfaces have been explored in several operating systems, including Nemesis [16]. Like the exokernel approach [20] and SPIN [7], Nemesis enables applications to perform as much of the processing as possible on their own using application-specific policy, supported by a set of operating system abstractions different from those in UNIX. Whereas Nemesis treats local network-interface bandwidth as the resource to be managed, we take a more end-to-end approach of discovering the end-to-end performance to different end-hosts, enabling sharing across common network paths. Furthermore, the API exported by Nemesis is useful for applications that can make resource reservations, while the CM API provides information about network conditions. Some "web switches" [?] provide traffic shaping and QoS based upon application information, but do not provide integrated flow management or feedback to the applications creating the data.

Multiple concurrent streams can cause problems for TCP congestion control. First, the ensemble of flows probes more aggressively for bandwidth than a single flow. Second, upon experiencing congestion along the path, only a subset of the connections usually reduce their window. Third, these flows do not share any information between each other. While we propose a gen-

eral solution to these problems, application-specific solutions have been proposed in the literature. Of particular importance are approaches that multiplex several logically distinct streams onto a single TCP connection at the application level, including Persistent-connection HTTP (P-HTTP [28], part of HTTP/1.1 [12]), the Session Control Protocol (SCP) [39], and the MUX protocol [15]. Unfortunately, these solutions suffer from two important drawbacks. First, because they are application-specific, they require each class of applications (Web, real-time streams, file transfers, etc.) to re-implement much of the same machinery. Second, they cause an undesirable coupling between logically different streams: if packets belonging to one stream are lost, another stream could stall even if none of its packets are lost because of the in-order "linear" delivery forced by TCP. Independent data units belonging to different streams are no longer independently processible and the parallelism of downloads is often lost.

7 Conclusion

The CM system enables applications to obtain an unprecedented degree of control over what they can do in response to different network conditions. It incorporates robust congestion control algorithms, freeing each application from having to re-implement them. It exposes a rich API that allows applications to adapt their transmissions at a fine-grained level, and allows the kernel and applications to integrate congestion information across flows.

Our evaluation of the CM implementation shows that the callback interface is effective for a variety of applications, and does not unduly burden the programmer with restrictive interfaces. From a performance standpoint, the CM itself imposes very little overhead; that which remains is mostly due to the unoptimized nature of our implementation. The architecture of programs implemented using UDP imposes some additional overhead, but the cost of using the CM after this architectural conversion is quite small.

Many systems exist to deliver content over the Internet using TCP or home-grown UDP protocols. We believe that by providing an accessible, robust framework for congestion control and adaptation, the Congestion Manager can help improve both the implementation and performance of these systems.

The Congestion Manager implementation for Linux is available from our web page, <http://nms.lcs.mit.edu/projects/cm/>.

Acknowledgements

We would like to thank Suchitra Raman and Alex Snoren for their helpful feedback and suggestions, and the Flux research group at the University of Utah for pro-

viding their network testbed³. Finally, we would like to thank our shepherd, Peter Druschel, and the anonymous reviewers for their numerous helpful comments. This work was supported by grants from DARPA (Grant No. MDA972-99-1-0014), NSF, IBM, Intel, and NTT Corporation.

References

- [1] BADRINATH, B. R., AND WELLING, G. Event Delivery Abstraction for Mobile Computing. Tech. Rep. LCSR-TR-242, Rutgers University, 1995.
- [2] BALAKRISHNAN, H., PADMANABHAN, V. N., SESHAN, S., STEMM, M., AND KATZ, R. TCP Behavior of a Busy Web Server: Analysis and Improvements. In *Proc. IEEE INFOCOM* (San Francisco, CA, Mar. 1998).
- [3] BALAKRISHNAN, H., RAHUL, H. S., AND SESHAN, S. An Integrated Congestion Management Architecture for Internet Hosts. In *Proc. ACM SIGCOMM* (Sep 1999).
- [4] BANGA, G., MOGUL, J. C., AND DRUSCHEL, P. A scalable and explicit event delivery mechanism for unix. In *Proc. of the USENIX 1999 Annual Technical Conference* (June 1999).
- [5] BARRERA, J. S. A fast Mach network IPC implementation. In *Proc. of the Second USENIX Mach Symposium* (Nov. 1991), pp. 1-12.
- [6] BERNERS-LEE, T., FIELDING, R., AND FRYSTYK, H. *Hypertext Transfer Protocol-HTTP/1.0*. Internet Engineering Task Force, May 1996. RFC 1945.
- [7] BERSHAD, B. N., SAVAGE, S., PARDYAK, P., SIRER, E. G., FIUCZYNSKI, M. E., BECKER, D., CHAMBERS, C., AND EGGERS, S. Extensibility, safety, and performance in the SPIN operating system. pp. 267-284.
- [8] BRADEN, B., CLARK, D., CROWCROFT, J., DAVIE, B., DEERING, S., ESTRIN, D., FLOYD, S., JACOBSON, V., MINSHALL, G., PARTRIDGE, C., PETERSON, L., RAMAKRISHNAN, K., SHENKER, S., WROCLAWSKI, J., AND ZHANG, L. *Recommendations on Queue Management and Congestion Avoidance in the Internet*. Internet Engineering Task Force, Apr 1998. RFC 2309.
- [9] CLARK, D. The Structuring of Systems Using Upcalls. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles (SOSP '85)* (Dec. 1985), pp. 171-180.
- [10] CLARK, D. The Design Philosophy of the DARPA Internet Protocols. In *Proc. ACM SIGCOMM* (Aug. 1988).
- [11] CLARK, D., AND TENNENHOUSE, D. Architectural Consideration for a New Generation of Protocols. In *Proc. ACM SIGCOMM* (September 1990).

³ Supported by NSF grant ANI-00-82493, DARPA/AFRL grant F30602-99-1-0503 and Cisco.

- [12] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., AND BERNERS-LEE, T. *Hypertext Transfer Protocol—HTTP/1.1*. Internet Engineering Task Force, Jan 1997. RFC 2068.
- [13] FLOYD, S., AND FALL, K. Promoting the Use of End-to-End Congestion Control in the Internet. *IEEE/ACM Trans. on Networking* 7, 4 (Aug. 1999).
- [14] FLOYD, S., JACOBSON, V., MCCANNE, S., LIU, C. G., AND ZHANG, L. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. In *Proc. ACM SIGCOMM* (Sept. 1995).
- [15] GETTYS, J. MUX protocol specification, WD-MUX-961023. <http://www.w3.org/pub/WWW/Protocols/MUX/WD-mux-961023.html>, 1996.
- [16] I. LESLIE AND D. MCAULEY AND R. BLACK AND T. ROSCOE AND P. BARHAM AND D. EVERS AND R. FAIRBAIRNS AND E. HYDEN. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications* 14, 7 (September 1996), 1280–1297.
- [17] INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC. *IEEE Standard for Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Programming Interface (API) — Amendment 1: Realtime Extension [C Language]*, 1994. Std 1003.1b-1993.
- [18] JACOBSON, V. Congestion Avoidance and Control. In *Proc. ACM SIGCOMM* (Aug 1988).
- [19] JACOBSON, V., BRADEN, R., AND BORMAN, D. *TCP Extensions for High Performance*. Internet Engineering Task Force, May 1992. RFC 1323.
- [20] KAASHOEK, M. F., ENGLER, D. R., GANGER, G. R., BRICEÑO, H. M., HUNT, R., MAZIÈRES, D., PINCKNEY, T., GRIMM, R., JANNOTTI, J., AND MACKENZIE, K. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)* (Saint-Malô, France, October 1997), pp. 52–65.
- [21] KARN, P., AND PARTRIDGE, C. Improving Round-Trip Time Estimates in Reliable Transport Protocols. *ACM Transactions on Computer Systems* 9, 4 (Nov. 1991), 364–373.
- [22] LEPREAU, J., ALFELD, C., ANDERSEN, D., AND MAREN, K. V. A large-scale network testbed. Unpublished, in 1999 SIGCOMM works-in-progress. <http://www.cs.utah.edu/flux/testbed/>, Sept. 1999.
- [23] The MASH Project Home Page. <http://www-mash.cs.berkeley.edu/mash/>, 1999.
- [24] MCCANNE, S., JACOBSON, V., AND VETTERLI, M. Receiver-driven Layered Multicast. In *Proc ACM SIGCOMM* (Aug. 1996).
- [25] Microsoft Windows Media Player. <http://www.microsoft.com/windows/mediaplayer/>.
- [26] NOBLE, B., SATYANARAYANAN, M., NARAYANAN, D., TILTON, J., FLINN, J., AND WALKER, K. Agile Application-Aware Adaptation for Mobility. In *Proc. 16th ACM Symposium on Operating Systems Principles* (Oct 1997).
- [27] PADMANABHAN, V. *Addressing the Challenges of Web Data Transport*. PhD thesis, Univ. of California, Berkeley, Sep 1998.
- [28] PADMANABHAN, V. N., AND MOGUL, J. C. Improving HTTP Latency. In *Proc. Second International WWW Conference* (Oct. 1994).
- [29] POSTEL, J. B. *User Datagram Protocol*. Internet Engineering Task Force, August 1980. RFC 768.
- [30] POSTEL, J. B. *Transmission Control Protocol*. Internet Engineering Task Force, September 1981. RFC 793.
- [31] QUINN, B., AND SHIUTE, D. *Windows Sockets Network Programming*. Addison-Wesley, Jan. 1999.
- [32] RAMAKRISHNAN, K., AND FLOYD, S. *A Proposal to Add Explicit Congestion Notification (ECN) to IP*. Internet Engineering Task Force, Jan 1999. RFC 2481.
- [33] RAMAN, S., AND MCCANNE, S. Scalable Data Naming for Application Level Framing in Reliable Multicast. In *Proc. ACM Multimedia* (Sept. 1998).
- [34] Real Networks. <http://www.real.com/>.
- [35] REJAIE, R., HANDLEY, M., AND ESTRIN, D. RAP: An End-to-end Rate-based Congestion Control Mechanism for Realtime Streams in the Internet. In *Proc. IEEE INFOCOM* (March 1999).
- [36] SAVAGE, S., CARDWELL, N., AND ANDERSON, T. The Case for Informed Transport Protocols. In *Proc. 7th Workshop on Hot Topics in Operating Systems (HotOS VII)* (Mar 1999).
- [37] SAVAGE, S., CARDWELL, N., WETHERALL, D., AND ANDERSON, T. TCP Congestion Control with a Misbehaving Receiver. In *ACM Computer Comm. Review* (Oct 1999).
- [38] SCHULZKRINNE, H., CASNER, S., FREDERICK, R., AND JACOBSON, V. *RTP: A Transport Protocol for Real-Time Applications*. Internet Engineering Task Force, Jan 1996. RFC 1889.
- [39] SPERO, S. Session Control Protocol (SCP). <http://www.w3.org/pub/WWW/Protocols/HTTP-NG/http-ng-scp.html>, 1996.
- [40] STEVENS, W. R. *TCP/IP Illustrated, Volume 1*. Addison-Wesley, Reading, MA, Nov 1994.
- [41] TAN, W., AND ZAKHOR, A. Real-time Internet Video Using Error Resilient Scalable Compression and TCP-friendly Transport Protocol. *IEEE Trans. on Multimedia* (May 1999).
- [42] TOUCH, J. *TCP Control Block Interdependence*. Internet Engineering Task Force, April 1997. RFC 2140.

Operating System Management of MEMS-based Storage Devices

John Linwood Griffin, Steven W. Schlosser,
Gregory R. Ganger, David F. Nagle
Carnegie Mellon University

Abstract

MEMS-based storage devices promise significant performance, reliability, and power improvements relative to disk drives. This paper compares and contrasts these two storage technologies and explores how the physical characteristics of MEMS-based storage devices change four aspects of operating system (OS) management: request scheduling, data placement, failure management, and power conservation. Straightforward adaptations of existing disk request scheduling algorithms are found to be appropriate for MEMS-based storage devices. A new bipartite data placement scheme is shown to better match these devices' novel mechanical positioning characteristics. With aggressive internal redundancy, MEMS-based storage devices can mask and tolerate failure modes that halt operation or cause data loss for disks. In addition, MEMS-based storage devices simplify power management because the devices can be stopped and started rapidly.

1 Introduction

Decades of research and experience have provided operating system builders with a healthy understanding of how to manage disk drives and their role in storage systems. This management includes achieving acceptable performance despite relatively time-consuming mechanical positioning delays, dealing with transient and permanent hardware problems so as to achieve high degrees of data survivability and availability, and minimizing power dissipation in battery-powered mobile environments. To address these issues, a wide array of OS techniques are used, including request scheduling, data layout, prefetching, caching, block remapping, data replication, and device spin-down. Given the prevalence and complexity of disks, most of these techniques have been specifically tuned to their physical characteristics.

When other devices (e.g., magnetic tape or Flash RAM) are used in place of disks, the characteristics of the problems change. Putting new devices behind

a disk-like interface is generally sufficient to achieve a working system. However, OS management techniques must be tuned to a particular device's characteristics to achieve the best performance, reliability, and lifetimes. For example, request scheduling techniques are much less important for RAM-based storage devices than for disks, since location-dependent mechanical delays are not involved. Likewise, locality-enhancing block layouts such as cylinder groups [18], extents [19], and log-structuring [24] are not as beneficial. However, log-structured file systems with idle-time cleaning can increase both performance and device lifetimes of Flash RAM storage devices with large erasure units [5].

Microelectromechanical systems (MEMS)-based storage is an exciting new technology that will soon be available in systems. MEMS are very small scale mechanical structures—on the order of 10–1000 μm —fabricated on the surface of silicon wafers [33]. These microstructures are created using photolithographic processes much like those used to manufacture other semiconductor devices (e.g., processors and memory) [7]. MEMS structures can be made to slide, bend, and deflect in response to electrostatic or electromagnetic forces from nearby actuators or from external forces in the environment. Using minute MEMS read/write heads, data bits can be stored in and retrieved from media coated on a small movable media sled [1, 11, 31]. Practical MEMS-based storage devices are the goal of major efforts at many research centers, including IBM Zurich Research Laboratory [31], Carnegie Mellon University [1], and Hewlett-Packard Laboratories [13].

Like disks, MEMS-based storage devices have unique mechanical and magnetic characteristics that merit specific OS techniques to manage performance, fault tolerance, and power consumption. For example, the mechanical positioning delays for MEMS-based storage devices depend on the initial and destination position and velocity of the media sled, just as disks' positioning times are dependent on the arm position and platter rotational offset. However, the

mechanical expressions that characterize sled motion differ from those describing disk platter and arm motion. These differences impact both request scheduling and data layout trade-offs. Similar examples exist for failure management and power conservation mechanisms. To assist designers of both MEMS-based storage devices and the systems that use them, an understanding of the options and trade-offs for OS management of these devices must be developed.

This paper takes a first step towards developing this understanding of OS management techniques for MEMS-based storage devices. It focuses on devices with the movable sled design that is being developed independently by several groups. With higher storage densities (260–720Gbit/in²) and lower random access times (less than 1 ms) than disks, these devices could play a significant role in future systems. After describing a disk-like view of these devices, we compare and contrast their characteristics with those of disks. Building on these comparisons, we explore options and implications for three major OS management issues: performance (specifically, request scheduling and block layout), failure management (media defects, device failures, and host crashes), and power conservation.

While these explorations are unlikely to represent the final word for OS management of these emerging devices, we believe that several of our high-level results will remain valid:

- Disk scheduling algorithms can be easily adapted to MEMS-based storage devices, improving performance much like they do for disks.
- Disk layout techniques can be adapted usefully, but the Cartesian movement of the sled (instead of the rotational motion of disks) allows further refinement of layouts.
- Striping of data and error correcting codes (ECC) across tips can greatly increase a device's tolerance to media, tip, and electronics faults; in fact, many faults that would halt operation or cause data loss in disks can be masked and tolerated in MEMS-based storage devices.
- OS power conservation is much simpler for MEMS-based storage devices. In particular, miniaturization and lack of rotation enable rapid transition between power-save and active modes, obviating the need for complex idle-time prediction and maximization algorithms.

The remainder of this paper is organized as follows. Section 2 describes MEMS-based storage devices, focusing on how they are similar to and different from magnetic disks. Section 3 describes our experimental setup, including the simulator and the workloads used. Section 4 evaluates request scheduling algorithms for MEMS-based storage devices. Section 5 explores data layout options. Section 6 describes approaches to fault management within and among MEMS-based storage devices. Section 7 discusses other device characteristics impacting the OS. Section 8 summarizes this paper's contributions.

2 MEMS-based storage devices

This section describes a MEMS-based storage device and compares and contrasts its characteristics with those of conventional disk drives. The description, which follows that of Reference [11], maps these devices onto a disk-like metaphor appropriate to their physical and operational characteristics.

2.1 Basic device description

MEMS-based storage devices can use the same basic magnetic recording technologies as disks to read and write data on the media. However, because it is difficult to build reliable rotating components in silicon, MEMS-based storage devices are unlikely to use rotating platters. Instead, most current designs contain a movable sled coated with magnetic media. This *media sled* is spring-mounted above a two-dimensional array of fixed read/write heads (*probe tips*) and can be pulled in the X and Y dimensions by electrostatic actuators along each edge. To access data, the media sled is first pulled to a specific location (x,y displacement). When this seek is complete, the sled moves in the Y dimension while the probe tips access the media. Note that the probe tips remain stationary—except for minute X and Z dimension movements to adjust for surface variation and skewed tracks—while the sled moves. In contrast, rotating platters and actuated read/write heads share the task of positioning in disks. Figures 1 and 2 illustrate this MEMS-based storage device design.

As a concrete example, the footprint of one MEMS-based storage device design is 196 mm², with 64 mm² of usable media area and 6400 probe tips [1]. Dividing the media into bit cells of 40×40 nm, and accounting for an ECC and encoding overhead of 2 bits per byte, this design has a formatted capacity of 3.2 GB/device. Note the square nature of the bit cells, which is not the case in conventional disk drives. With minute probe tips and vertical record-

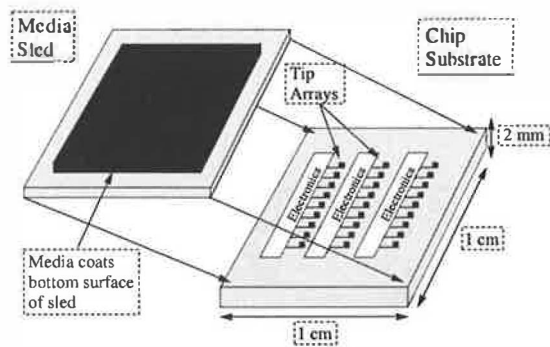


Figure 1: *Components of a MEMS-based storage device.* The media sled is suspended above an array of probe tips. The sled moves small distances along the *X* and *Y* axes, allowing the stationary tips to address the media.

ing, bits stored on these devices can have a 1-to-1 aspect ratio, resulting in areal densities 15–30 times greater than those of disks. However, per-device capacities are lower because individual MEMS-based storage devices are much smaller than disks. Because the mechanically-positioned MEMS components have much smaller masses than corresponding disk parts, their random access times are in the hundreds of microseconds. For the default device parameters in this paper, the average random 4 KB access time is 703 μ s.

2.2 Low-level data layout

The storage media on the sled is divided into rectangular regions as shown in Figure 3. Each region contains $M \times N$ bits (e.g., 2500×2500) and is accessible by exactly one probe tip; the number of regions on the media equals the number of probe tips. Each term in the nomenclature below is defined both in the text and visually in Figure 4.

Cylinders. Drawing on the analogy to disk terminology, we refer to a *cylinder* as the set of all bits with identical *x* offset within a region (i.e., at identical sled displacement in *X*). In other words, a cylinder consists of all bits accessible by all tips when the sled moves only in the *Y* dimension, remaining immobile in the *X* dimension. Cylinder 1 is highlighted in Figure 4 as the four circled columns of bits. This definition parallels that of disk cylinders, which consist of all bits accessible by all heads while the arm remains immobile. There are *M* cylinders per sled. In our default model, each sled has 2500 cylinders that each hold 1350 KB of data.

Tracks. A MEMS-based storage device might have 6400 tips underneath its media sled; however, due

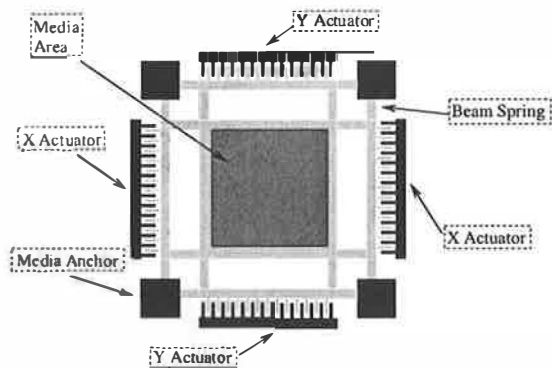


Figure 2: *The movable media sled.* The actuators, spring suspension, and the media sled are shown. Anchored regions are solid and the movable structure is shaded grey.

to power and heat considerations it is unlikely that all 6400 tips can be *active* (accessing data) concurrently. We expect to be able to activate 200–2000 tips at a time. To account for this limitation, we divide cylinders into tracks. A *track* consists of all bits within a cylinder that can be read by a group of concurrently active tips. The sled in Figure 4 has sixteen tips (one per region; not all tips are shown), of which up to four can be concurrently active—each cylinder therefore has four tracks. Track 0 of cylinder 1 is highlighted in the figure as the leftmost circled column of bits. Note again the parallel with disks, where a track consists of all bits within a cylinder accessible by a single active head. In our default model, each sled has 6400 tips and 1280 concurrently active tips, so each cylinder contains 5 tracks that each hold 270 KB of data. Excluding positioning time, accessing an entire track takes 3.47 ms.

Sectors. Continuing the disk analogy, tracks are divided into sectors. Instead of having each active tip read or write an entire vertical column of *N* bits, each tip accesses only 90 bits at a time—10 bits of servo/tracking information and 80 data bits (8 encoded data bytes). Each 80-data-bit group forms an 8-byte *sector*, which is the smallest individually accessible unit of data on our MEMS-based storage device. Each track in Figure 4 contains 12 sectors (3 per tip). These sectors parallel the partitioning of disk tracks into sectors, with three notable differences. First, disk sectors contain more data (e.g., 512 bytes vs. 8 bytes). Second, MEMS-based storage devices can access multiple sectors concurrently: Figure 4 shows the four active tips accessing sectors 4, 5, 6, and 7. Third, MEMS-based storage devices can support *bidirectional access*, meaning that a data sector can be accessed in either the +*Y* or

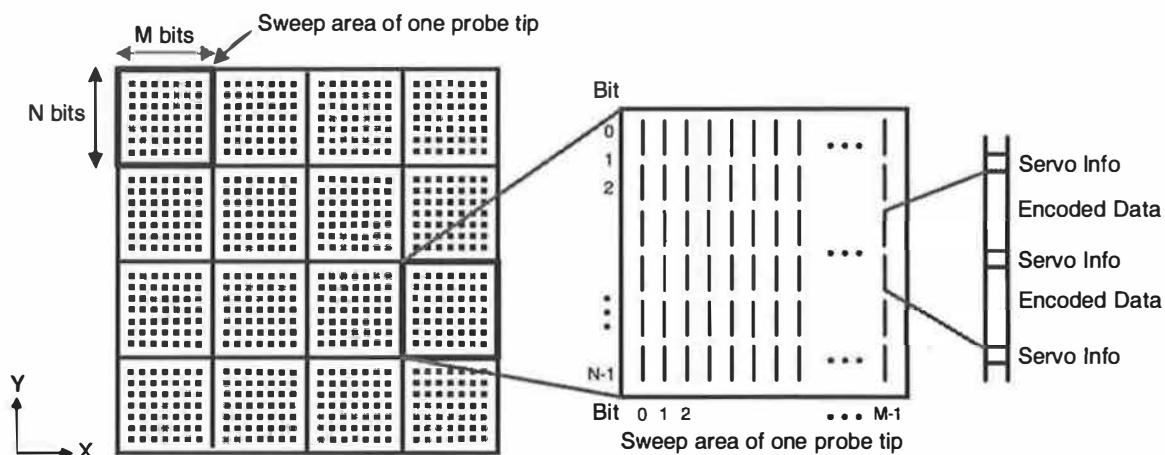


Figure 3: **Data organization on MEMS-based storage devices.** The illustration depicts a small portion of the magnetic media sled. Each small rectangle outlines the media area accessible by a single probe tip, with a total of 16 tip regions shown. A full device contains thousands of tips and tip regions. Each region stores $M \times N$ bits, organized into M vertical columns of N bits, alternating between servo/tracking information (10 bits) and data (80 bits = 8 encoded data bytes). To read or write data, the media sled passes over the tips in the $\pm Y$ directions while the tips access the media.

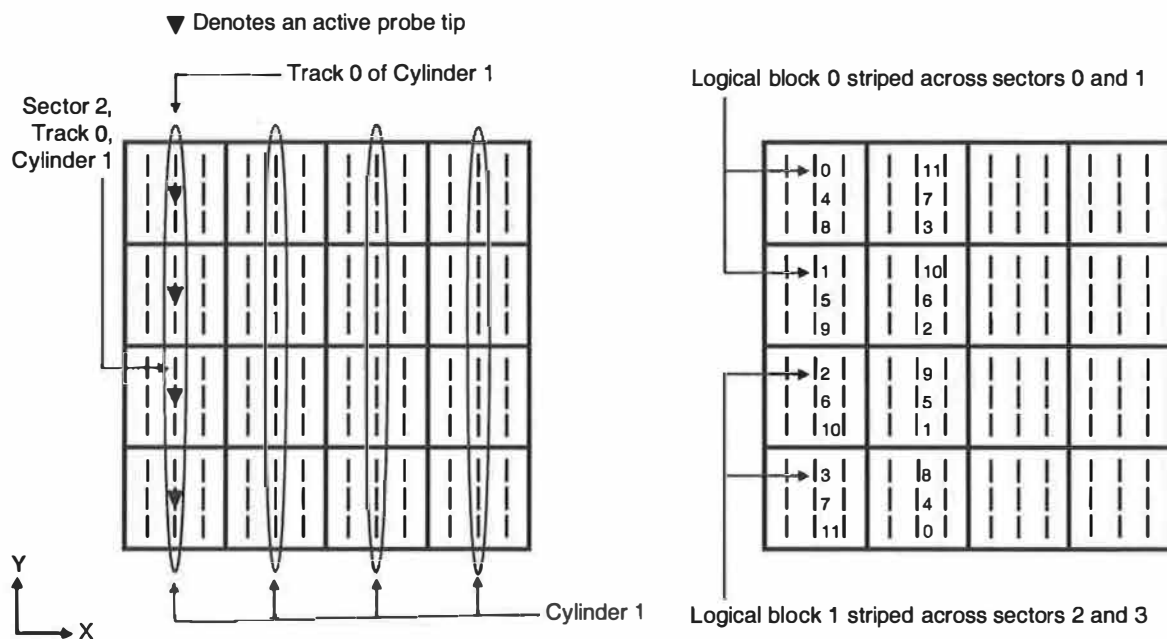


Figure 4: **Cylinders, tracks, sectors, and logical blocks.** This example shows a MEMS-based storage device with 16 tips and $M \times N = 3 \times 280$. A "cylinder" is defined as all data at the same x offset within all regions; cylinder 1 is indicated by the four circled columns of bits. Each cylinder is divided into 4 "tracks" of 1080 bits, where each track is composed of four tips accessing 280 bits each. Each track is divided into 12 "sectors" of 80 bits each, with 10 bits of servo/tracking information between adjacent sectors and at the top and bottom of each track. (There are nine sectors in each tip region in this example.) Finally, sectors are grouped together in pairs to form "logical blocks" of 16 bytes each. Sequential sector and logical block numbering are shown on the right. These definitions are discussed in detail in Section 2.2.

–Y direction. In our default model, each track is composed of 34,560 sectors of 8 bytes each, of which up to 1280 sectors can be accessed concurrently. Excluding positioning time, each 1280 sector (10 KB) access takes 0.129 ms.

Logical blocks. For the experiments in this paper, we combine groups of 64 sectors into SCSI-like *logical blocks* of 512 bytes each. Each logical block is therefore striped across 64 tips, and up to 20 logical blocks can be accessed concurrently ($1280 \div 64 = 20$). During a request, only those logical blocks needed to satisfy the request and any firmware-directed prefetching are accessed; unused tips remain inactive to conserve power.

2.3 Media access characteristics

Media access requires constant sled velocity in the Y dimension and zero velocity in the X dimension. The Y dimension *access speed* is a design parameter and is determined by the per-tip read and write rates, the bit cell width, and the sled actuator force. Although read and write data rates could differ, tractable control logic is expected to dictate a single access velocity in early MEMS-based storage devices. In our default model, the access speed is 28 mm/s and the corresponding per-tip data rate is 0.7 Mbit/s.

Positioning the sled for read or write involves several mechanical and electrical actions. To seek to a sector, the appropriate probe tips must be activated (to access the servo information and then the data), the sled must be positioned at the correct x,y displacement, and the sled must be moving at the correct velocity for access. Whenever the sled seeks in the X dimension—i.e., the destination cylinder differs from the starting cylinder—extra *settling time* must be taken into account because the spring-sled system oscillates in X after each cylinder-to-cylinder seek. Because this oscillation is large enough to cause off-track interference, a closed loop settling phase is used to damp the oscillation. To the first order, this active damping is expected to require a constant amount of time. Although slightly longer settling times may ultimately be needed for writes, as is the case with disks, we currently assume that the settling time is the same for both read and write requests. Settling time is not a factor in Y dimension seeks because the oscillations in Y are subsumed by the large Y dimension access velocity and can be tolerated by the read/write channel.

As the sled is moved away from zero displacement, the springs apply a restoring force toward the sled's rest position. These spring forces can either improve or degrade positioning time (by affecting the effec-

tive actuator force), depending on the sled displacement and direction of motion. This force is parameterized in our simulator by the *spring factor*—the ratio of the maximum spring force to the maximum actuator force. A spring factor of 75% means that the springs pull toward the center with 75% of the maximum actuator force when the sled is at full displacement. The spring force decreases linearly to 0% as sled displacement approaches zero. The spring restoring force makes the acceleration of the sled a function of instantaneous sled position. In general, the spring forces tend to degrade the seek time of short seeks and improve the seek time of long seeks [11].

Large transfers may require that data from multiple tracks or cylinders be accessed. To switch tracks during large transfers, the sled switches which tips are active and performs a *turnaround*, using the actuators to reverse the sled's velocity (e.g., from +28 mm/s to –28 mm/s). The turnaround time is expected to dominate any additional activity, such as the time to activate the next set of active tips, during both track and cylinder switches. One or two turnarounds are necessary for any seek in which the sled is moving in the wrong direction—away from the sector to be accessed—before or after the seek.

2.4 Comparison to conventional disks

Although MEMS-based storage devices involve some radically different technologies from disks, they share enough fundamental similarity for a disk-like model to be a sensible starting point. Like disks, MEMS-based storage devices stream data at a high rate and suffer a substantial distance-dependent positioning time delay before each nonsequential access. In fact, although MEMS-based storage devices are much faster, they have ratios of request throughput to data bandwidth similar to those of disks from the early 1990s. Some values of the ratio, γ , of request service rate (IO/s) to streaming bandwidth (MB/s) for some recent disks include $\gamma = 26$ (1989) for the CDC Wren-IV [21], $\gamma = 17$ (1993) [12], and $\gamma = 5.2$ (1999) for the Quantum Atlas 10K [22]. γ for disks continue to drop over time as bandwidth improves faster than mechanical positioning times. In comparison, the MEMS-based storage device in this paper yields $\gamma = 19$ ($1422 \text{ IO/s} \div 76 \text{ MB/s}$), comparable to disks within the last decade. Also, although many probe tips access the media in parallel, they are all limited to accessing the same relative x,y offset within a region at any given point in time—recall that the media sled moves freely while the probe tips remain relatively fixed. Thus, the probe tip paral-

lelism provides greater data rates but not concurrent, independent accesses. There are alternative physical device designs that would support greater access concurrency and lower positioning times, but at substantial cost in capacity [11].

The remainder of this section enumerates a number of relevant similarities and differences between MEMS-based storage devices and conventional disk drives. With each item, we also discuss consequences for device management issues and techniques.

Mechanical positioning. Both disks and MEMS-based storage devices have two main components of positioning time for each request: seek and rotation for disks, X and Y dimension seeks for MEMS-based storage devices. The major difference is that the disk components are independent (i.e., desired sectors rotate past the read/write head periodically, independent of when seeks complete), whereas the two components are explicitly done in parallel for MEMS-based storage devices. As a result, total positioning time for MEMS-based storage equals the greater of the X and Y seek times, making the lesser time irrelevant. The effect of this overlap on request scheduling is discussed in Section 4.2.

Settling time. For both disks and MEMS-based storage devices, it is necessary for read/write heads to settle over the desired track after a seek. Settling time for disks is a relatively small component of most seek times (0.5 ms of 1–15 ms seeks). However, settling time for MEMS-based storage devices is expected to be a relatively substantial component of seek time (0.2 ms of 0.2–0.8 ms seeks). Because the settling time is generally constant, this has the effect of making seek times more constant, which in turn could reduce (but not eliminate) the benefit of both request scheduling and data placement. Section 4.3 discusses this issue.

Logical-to-physical mappings. As with disks, we expect the lowest-level mapping of logical block numbers (*LBNs*) to physical locations to be straightforward and optimized for sequential access; this will be best for legacy systems that use these new devices as disk replacements. Such a sequentially optimized mapping scheme fits disk terminology and has some similar characteristics. Nonetheless, the physical differences will make data placement decisions (mapping of file or database blocks to *LBNs*) an interesting topic. Section 5 discusses this issue.

Seek time vs. seek distance. For disks, seek times are relatively constant functions of the seek distance, independent of the start cylinder and direction of seek. Because of the spring restoring

forces, this is not true of MEMS-based storage devices. Short seeks near the edges take longer than they do near the center (as discussed in Section 5). Also, turnarounds near the edges take either less time or more, depending on the direction of sled motion. As a result, seek-reducing request scheduling algorithms [34] may not achieve their best performance if they look only at distances between *LBNs* as they can with disks.

Recording density. Some MEMS-based storage devices use the same basic magnetic recording technologies as disks [1]. Thus, the same types of fabrication and grown media defects can be expected. However, because of the much higher bit densities of MEMS-based storage devices, each such media defect will affect a much larger number of bits. This is one of the fault management issues discussed in Section 6.1.

Numbers of mechanical components. MEMS-based storage devices have many more distinct mechanical parts than disks. Although their very small movements make them more robust than the large disk mechanics, their sheer number makes it much more likely that some number of them will break. In fact, manufacturing yields may dictate that the devices operate with some number of broken mechanical components. Section 6.1 discusses this issue.

Concurrent read/write heads. Because it is difficult and expensive for drive manufacturers to enable parallel activity, most modern disk drives use only one read/write head at a time for data access. Even drives that do support parallel activity are limited to only 2–20 heads. On the other hand, MEMS-based storage devices (with their per-tip actuation and control components) could theoretically use all of their probe tips concurrently. Even after power and heat considerations, hundreds or thousands of concurrently active probe tips is a realistic expectation. This parallelism increases media bandwidth and offers opportunities for improved reliability. Section 6.1 discusses the latter.

Control over mechanical movements. Unlike disks, which rotate at constant velocity independent of ongoing accesses, the mechanical movements of MEMS-based storage devices can be explicitly controlled. As a result, access patterns that suffer significantly from independent rotation can be better served. The best example of this is repeated access to the same block, as often occurs for synchronous metadata updates or read-modify-write sequences. This difference is discussed in Section 6.2.

Startup activities. Like disks, MEMS-based storage devices will require some time to ready themselves for media accesses when powered up. However, because of the size of their mechanical structures and their lack of rotation, the time and power required for startup will be much less than for disks. The consequences of this fact for both availability (Section 6.3) and power management (Section 7) are discussed in this paper.

Drive-side management. As with disks, management functionality will be split between host OSes and device OSes (firmware). Over the years, increasing amounts of functionality have shifted into disk firmware, enabling a variety of portability, reliability, mobility, performance, and scalability enhancements. We expect a similar trend with MEMS-based storage devices, whose silicon implementations offer the possibility of direct integration of storage with computational logic.

Speed-matching buffers. As with disks, MEMS-based storage devices access the media as the sled moves past the probe tips at a fixed rate. Since this rate rarely matches that of the external interface, speed-matching buffers are important. Further, because sequential request streams are important aspects of many real systems, these speed-matching buffers will play an important role in prefetching and then caching of sequential LBNs. Also, as with disks, most block reuse will be captured by larger host memory caches instead of in the device cache.

Sectors per track. Disk media is organized as a series of concentric circles, with outer circles having larger circumferences than inner circles. This fact led disk manufacturers to use banded (zoned) recording in place of a constant bit-per-track scheme in order to increase storage density and bandwidth. For example, banded recording results in a 3:2 ratio between the number of sectors on the outermost (334 sectors) and innermost (229 sectors) tracks in the Quantum Atlas 10K [8]. Because MEMS-based storage devices instead organize their media in fixed-size columns, there is no length difference between tracks and banded recording is not relevant. Therefore, block layout techniques that try to exploit banded recording will not provide benefit for these devices. On the other hand, for block layouts that try to consider track boundaries and block offsets within tracks, this uniformity (which was common in disks 10 or more years ago) will simplify or enable correct implementations. The subregioned layout described in Section 5 is an example of such a layout.

device capacity	3.2 GB
number of tips	6400
maximum concurrent tips	1280
sled acceleration	803.6 m/s ²
sled access speed	28 mm/s
constant settling time	0.22 ms
spring factor	75%
per-tip data rate	0.7 Mbit/s
media bit cell size	40×40 nm
bits per tip region (M×N)	2500×2440
data encoding overhead	2 bits per byte
servo overhead per 8 bytes	10 bits (11%)
command processing overhead	0.2 ms/request
on-board cache memory	0 MB
external bus bandwidth	100 MB/s

Table 1: *Default MEMS-based storage device parameters. $N=2440$ in order to fit an integral number of 80-bit encoded sectors (with inter-sector servo) in each column of bits. The default model includes no on-board caching (or prefetching), but does assume speed-matching buffers between the tips and the external bus.*

3 Experimental setup

The experiments in this paper use the performance model for MEMS-based storage described in Reference [11], which includes all of the characteristics described above. Although it is not yet possible to validate the model against real devices, both the equations and the default parameters are the result of extensive discussions with groups that are designing and building MEMS-based storage devices [2, 3, 20]. We therefore believe that the model is sufficiently representative for the insights gained from experiments to be useful. Table 1 shows default parameters for the MEMS-based storage device simulator.

This performance model has been integrated into the DiskSim simulation environment [10] as a disk-like storage device accessed via a SCSI-like protocol. DiskSim provides an infrastructure for exercising the device model with various synthetic and trace-based workloads. DiskSim also includes a detailed, validated disk module that can accurately model a variety of real disks. For reference, some experiments use DiskSim’s disk module configured to emulate the Quantum Atlas 10K, one of the disks for which publicly available configuration parameters have been calibrated against real-world drives [8]. The Quantum Atlas 10K has a nominal rotation speed of 10,000 RPM, average seek time of 5.0 ms, streaming bandwidth of 17.3–25.2 MB/s, and average random single-sector access time of 8.5 ms [22].

Some of the experiments use a synthetically-generated workload that we refer to as the *Random* workload. For this workload, request inter-arrival times are drawn from an exponential distribution; the mean is varied to simulate a range of workloads. All other aspects of requests are independent: 67% are reads, 33% are writes, the request size distribution is exponential with a mean of 4 KB, and request starting locations are uniformly distributed across the device's capacity.

For more realistic workloads, we use two traces of real disk activity: the *TPC-C* trace and the *Cello* trace. The TPC-C trace comes from a TPC-C testbed, consisting of Microsoft SQL Server atop Windows NT. The hardware was a 300 MHz Intel Pentium II-based system with 128 MB of memory and a 1 GB test database striped across two Quantum Viking disk drives. The trace captures one hour of disk activity for TPC-C, and its characteristics are described in more detail in Reference [23]. The Cello trace comes from a Hewlett-Packard system running the HP-UX operating system. It captures disk activity from a server at HP Labs used for program development, simulation, mail, and news. While the total trace is actually two months in length, we report data for a single, day-long snapshot. This trace and its characteristics are described in detail in Reference [25]. When replaying the traces, each traced disk is replaced by a distinct simulated MEMS-based storage device.

As is often the case in trace-based studies, our simulated devices are newer and significantly faster than the disks used in the traced systems. To explore a range of workload intensities, we replicate an approach used in previous disk scheduling work [34]: we scale the traced inter-arrival times to produce a range of average inter-arrival times. When the scale factor is one, the request inter-arrival times match those of the trace. When the scale factor is two, the traced inter-arrival times are halved, doubling the average arrival rate.

4 Request scheduling

An important mechanism for improving disk efficiency is deliberate scheduling of pending requests. Request scheduling improves efficiency because positioning delays are dependent on the relative positions of the read/write head and the destination sector. The same is true of MEMS-based storage devices, whose seek times are dependent on the distance to be traveled. This section explores the impact of different scheduling algorithms on the performance of MEMS-based storage devices.

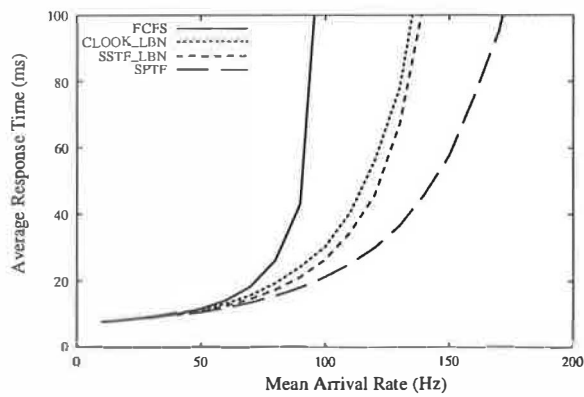
4.1 Disk scheduling algorithms

Many disk scheduling algorithms have been devised and studied over the years. Our comparisons focus on four. First, the simple *FCFS* (first-come, first-served) algorithm often results in suboptimal performance, but we include it for reference. The *SSTF* (shortest seek time first) algorithm was designed to select the request that will incur the smallest seek delay [4], but this is rarely the way it functions in practice. Instead, since few host OSes have the information needed to compute actual seek distances or predict seek times, most SSTF implementations use the difference between the last accessed LBN and the desired LBN as an approximation of seek time. This simplification works well for disk drives [34], and we label this algorithm as *SSTF_LBN*. The *CLOOK_LBN* (cyclical look) algorithm services requests in ascending LBN order, starting over with the lowest LBN when all requests are “behind” the most recent request [28]. The *SPTF* (shortest positioning time first) policy selects the request that will incur the smallest positioning delay [14, 29]. For disks, this algorithm differs from others in that it explicitly considers both seek time and rotational latency.

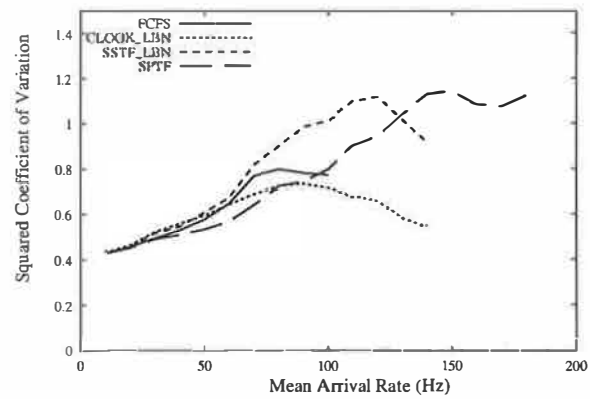
For reference, Figure 5 compares these four disk scheduling algorithms for the Atlas 10K disk drive and the Random workload (Section 3) with a range of request arrival rates. Two common metrics for evaluating disk scheduling algorithms are shown. First, the average response time (queue time plus service time) shows the effect on average performance. As expected, FCFS saturates well before the other algorithms as the workload increases. SSTF_LBN outperforms CLOOK_LBN, and SPTF outperforms all other schemes. Second, the squared coefficient of variation (σ^2/μ^2) is a metric of “fairness” (or starvation resistance) [30, 34]; lower values indicate better starvation resistance. As expected, CLOOK_LBN avoids the starvation effects that characterize the SSTF_LBN and SPTF algorithms. Although not shown here, age-weighted versions of these greedy algorithms can reduce request starvation without unduly reducing average case performance [14, 29].

4.2 MEMS-based storage scheduling

Existing disk scheduling algorithms can be adapted to MEMS-based storage devices once these devices are mapped onto a disk-like interface. Most algorithms, including SSTF_LBN and CLOOK_LBN, only use knowledge of LBNs and assume that differences between LBNs are reasonable approxima-



(a) Average response times



(b) Squared coefficients of variation (σ^2/μ^2)

Figure 5: *Comparison of scheduling algorithms for the Random workload on the Quantum Atlas 10K disk.*

tions of positioning times. SPTF, which addresses disk seeks and rotations, is a more interesting case. While MEMS-based storage devices do not have a rotational latency component, they do have two positioning time components: the X dimension seek and the Y dimension seek. As with disks, only one of these two components (seek time for disks; the X dimension seek for MEMS-based storage devices) is approximated well by a linear LBN space. Unlike disks, the two positioning components proceed in parallel, with the greater subsuming the lesser. The settling time delay makes most X dimension seek times larger than most Y dimension seek times. Although it should never be worse, SPTF will only be better than SSTF (which minimizes X movements, but ignores Y) when the Y component is frequently the larger.

Figure 6 shows how well these algorithms work for the default MEMS-based storage device on the Random workload with a range of request arrival rates. In terms of both performance and starvation resistance, the algorithms finish in the same order as for disks: SPTF provides the best performance and CLOOK_LBN provides the best starvation resistance. However, their performance relative to each other merits discussion. The difference between FCFS and the LBN-based algorithms (CLOOK_LBN and SSTF_LBN) is larger for MEMS-based storage devices because the seek time is a much larger component of the total service time. In particular, there is no subsequent rotational delay. Also, the average response time difference between CLOOK_LBN and SSTF_LBN is smaller for MEMS-based storage devices, because both algorithms reduce the X seek times into the range where X and Y seek times are comparable. Since neither addresses

Y seeks, the greediness of SSTF_LBN is less effective. SPTF obtains additional performance by addressing Y seeks.

Figures 7(a) and 7(b) show how the scheduling algorithms perform for the Cello and TPC-C workloads, respectively. The relative performance of the algorithms on the Cello trace is similar to the Random workload. The overall average response time for Cello is dominated by the busiest one of Cello's eight disks; some of the individual disks have differently shaped curves but still exhibit the same ordering among the algorithms. One noteworthy difference between TPC-C and Cello is that SPTF outperforms the other algorithms by a much larger margin than for TPC-C at high loads. This occurs because the scaled-up version of the workload includes many concurrently-pending requests with very small LBN distances between adjacent requests. LBN-based schemes do not have enough information to choose between such requests, often causing small (but expensive) X-dimension seeks. SPTF addresses this problem and therefore performs much better.

4.3 SPTF and settling time

Originally, we had expected SPTF to outperform the other algorithms by a greater margin for MEMS-based storage devices. Our investigations suggest that the value of SPTF scheduling is highly dependent upon the settling time component of X dimension seeks. With large settling times, X dimension seek times dominate Y dimension seek times, making SSTF_LBN match SPTF. With small settling times, Y dimension seek times are a more significant component. To illustrate this, Figure 8 compares the scheduling algorithms with the constant settling time set to zero and 0.44 ms (double the de-

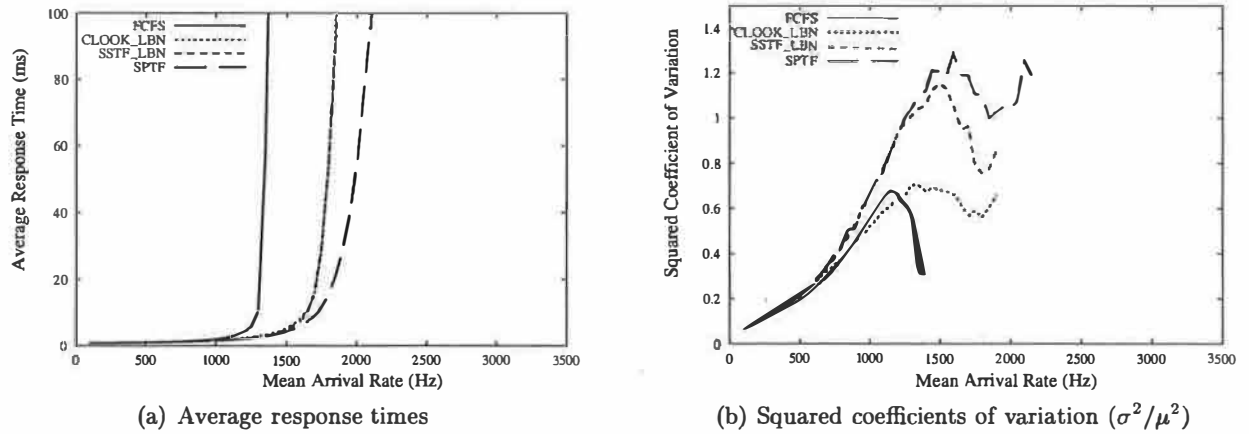


Figure 6: *Comparison of scheduling algorithms for the Random workload on the MEMS-based storage device. Note the scale of the X axis has increased by an order of magnitude relative to the graphs in Figure 5.*

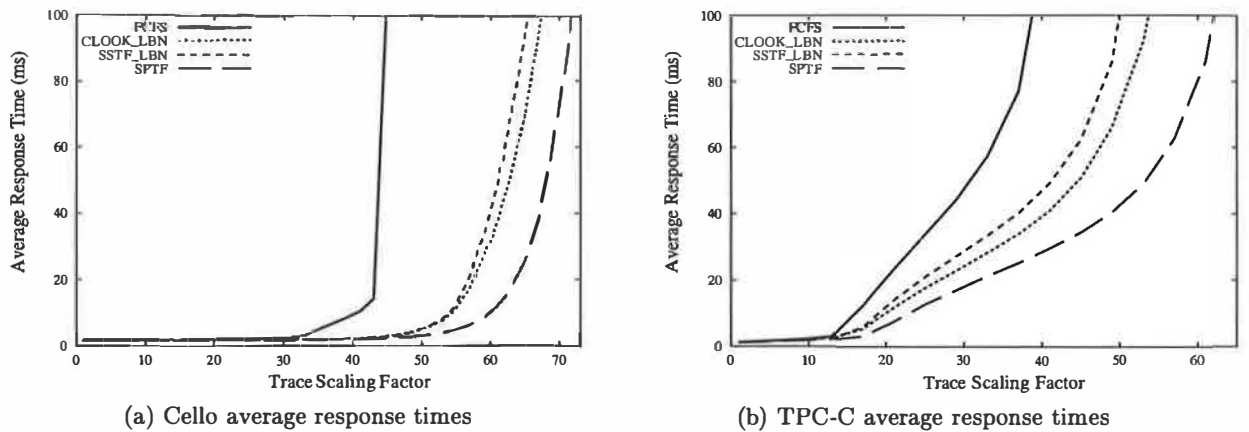


Figure 7: *Comparison of scheduling algorithms for the Cello and TPC-C workloads on the MEMS-based storage device.*

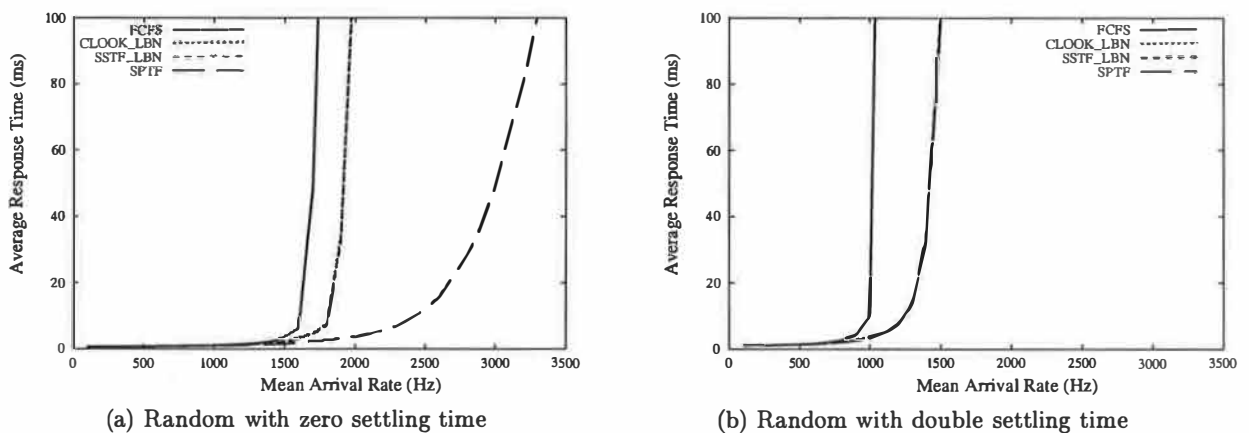


Figure 8: *Comparison of average performance of the Random workload for zero and double constant settling time on the MEMS-based storage device. These are in comparison to the default model (Random with constant settling time of 0.22 ms) shown in Figure 6(a). With no settling time, SPTF significantly outperforms CLOOK_LBN and SSTF_LBN. With the doubled settling time, CLOOK_LBN, SSTF_LBN, and SPTF are nearly identical.*

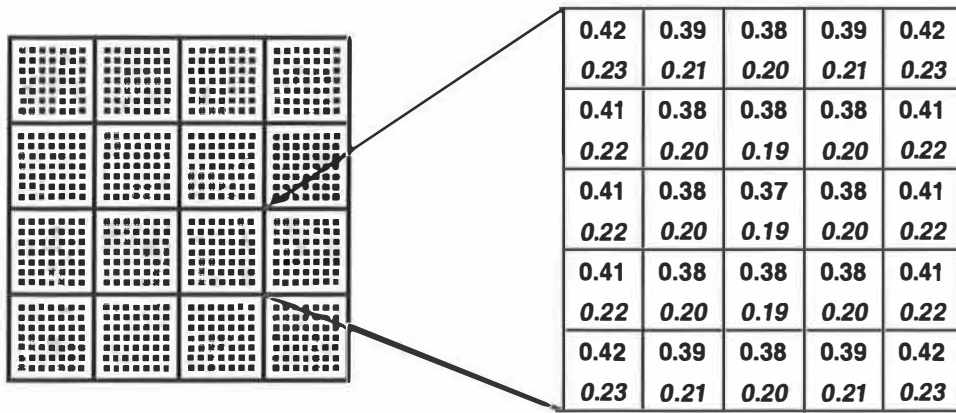


Figure 9: *Difference in request service time for subregion accesses.* This figure divides the region accessible by an individual probe tip into 25 subregions, each 500×500 bits. Each box shows the average request service time (in milliseconds) for random requests starting and ending inside that subregion. The upper numbers represent the service time when the default settling time is included in calculations; numbers in italics represent the service time for zero settling time. Note that the service time differs by 14–21% between the centermost and outermost subregions.

fault value). As expected, SSTF.LBN is very close to SPTF when the settling time is doubled. With zero settling time, SPTF outperforms the other algorithms by a large margin.

5 On-device data layout

Space allocation and data placement for disks continues to be a ripe topic of research. We expect the same to be true of MEMS-based storage devices. In this section, we discuss how the characteristics of MEMS-based storage positioning costs affect placement decisions for small local accesses and large sequential transfers. A bipartite layout is proposed and shown to have potential for improving performance.

5.1 Small, skewed accesses

As with disks, short distance seeks are faster than long distance seeks. Unlike disks, MEMS-based storage devices' spring restoring forces make the effective actuator force (and therefore sled positioning time) a function of location. Figure 9 shows the impact of spring forces for seeks inside different “subregions” of a single tip's media region. The spring forces increase with increasing sled displacement from the origin (viz., toward the outermost subregions in Figure 9), resulting in longer positioning times for short seeks. As a result, distance is not the only component to be considered when finding good placements for small, popular data items—offset relative to the center should also be considered.

5.2 Large, sequential transfers

Streaming media transfer rates for MEMS-based storage devices and disks are similar: 17.3–25.2MB/s for the Atlas 10K [22]; 75.9 MB/s for MEMS-based storage devices. Positioning times, however, are an order of magnitude shorter for MEMS-based storage devices than for disks. This makes positioning time relatively insignificant for large transfers (e.g., hundreds of sectors). Figure 10 shows the request service times for a 256 KB read with respect to the X distance between the initial and final sled positions. Requests traveling 1250 cylinders (e.g., from the sled origin to maximum sled displacement) incur only a 10% penalty. This lessens the importance of ensuring locality for data that will be accessed in large, sequential chunks. In contrast, seek distance is a significant issue with disks, where long seeks more than double the total service time for 256 KB requests.

5.3 A data placement scheme for MEMS-based storage devices

To take advantage of the above characteristics, we propose a 25-subregion bipartite layout scheme. Small data are placed in the centermost subregions; long, sequential streaming data are placed in outer subregions. Two layouts are tested: a five-by-five grid of subregions (Figure 9) and a simple columnar division of the LBN space into 25 columns (viz., column 0 contains cylinders 0–99, column 1 contains cylinders 100–199, etc.).

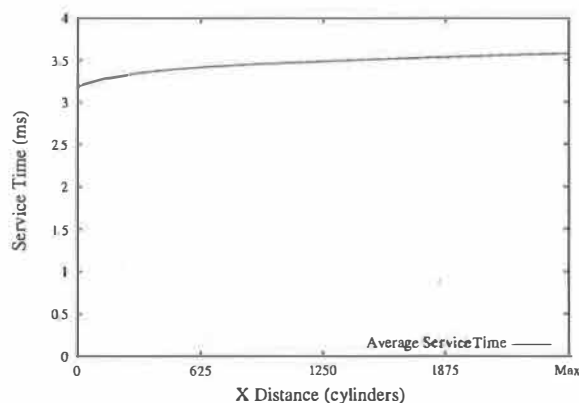


Figure 10: *Large (256 KB) request service time vs. X seek distance for MEMS-based storage devices.* Because the media access time is large relative to the positioning time, seeking the maximum distance in X increases the service time for large requests by only 12%.

We compare these layout schemes against the “organ pipe” layout [26, 32], an optimal disk-layout scheme, assuming no inter-request dependencies. In the organ pipe layout, the most frequently accessed files are placed in the centermost tracks of the disk. Files of decreasing popularity are distributed to either side of center, with the least frequently accessed files located closer to the innermost and outermost tracks. Although this scheme is optimal for disks, files must be periodically shuffled to maintain the frequency distribution. Further, the layout requires some state to be kept, indicating each file’s popularity.

To evaluate these layouts, we used a workload of 10,000 whole-file read requests whose sizes are drawn from the file size distribution reported in Reference [9]. In this size distribution, 78% of files are 8 KB or smaller, 4% are larger than 64 KB, and 0.25% are larger than 1 MB. For the subregioned and columnar layouts, the large files (larger than 8 KB) were mapped to the ten leftmost and ten rightmost subregions, while the small files (8 KB or less) were mapped to the centermost subregion. To conservatively avoid second-order locality within the large or small files, we assigned a random location to each request within either the large or the small subregions. For the organ pipe layout, we used an exponential distribution to determine file popularity, which was then used to place files.

Figure 11 shows that all three layout schemes achieve a 12–15% improvement in average access time over a simple random file layout. Subregioned and colum-

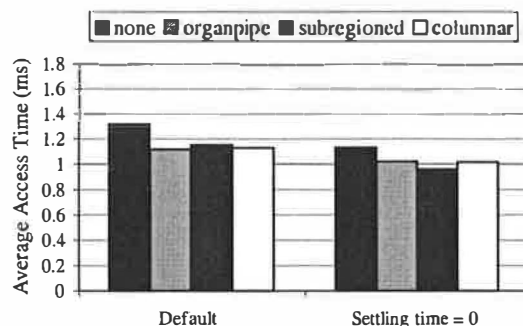


Figure 11: *Comparison of layout schemes for MEMS-based storage devices.* For the default device, the organ pipe, subregioned, and columnar layouts achieve a 12–15% performance improvement over a random layout. Further, for the “settling time = 0” case, the subregioned layout outperforms the others by an additional 12%. It is interesting to note that an optimal disk layout technique does not necessarily provide the best performance for MEMS-based storage.

nar layouts for MEMS-based storage devices match organ pipe, even with the conservative model and no need for keeping popularity data or periodically reshuffling files on the media. For the “no settling time” case, the subregioned layout provides the best performance as it addresses both X and Y.

6 Failure management

Fault tolerance and recoverability are significant considerations for storage systems. Although MEMS-based storage devices are not yet available, MEMS components have been built and tested for many years. Their miniature size and movements will make MEMS-based storage components less fragile than their disk counterparts [17]. Still, there will likely be more defective or failed parts in MEMS-based storage because of the large number of distinct components compared to disks and the fact that bad parts cannot be replaced before or during assembly.

Although failure management for MEMS-based storage devices will be similar to failure management for conventional disks, there are several important differences. One is that individual component failures must be made less likely to render a device inoperable than in disks. Another is that MEMS-based storage devices simplify some aspects of failure management—inter-device redundancy maintenance and device restart, for example. This section discusses three aspects of failure management: internal faults, device failures, and recoverability from system crashes.

6.1 Internal faults

The common failure modes for disk drives include recoverable failures (for example, media defects or seek errors) and non-recoverable failures (head crashes, motor or arm actuator failure, drive electronics or channel failure). MEMS-based storage devices have similar failure modes with analogous causes. However, the ability to incorporate multiple tips into failure tolerance schemes allows MEMS-based storage devices to mask most component failures, including many that would render a disk inoperable.

Specifically, powerful error correcting codes can be computed over data striped across multiple tips. In our default model, each 512 byte logical block and its ECC are striped across 64 tips. This ECC can include both a horizontal component (across tips) and a vertical component (within a single sector). The horizontal ECC can recover from missing sectors. The vertical ECC identifies sectors that should be treated as missing—with the effect of converting some large errors into erasures, which can more easily be handled by the horizontal ECC. This single mechanism addresses most internal failures that are recoverable.

Media defects. In disk drives, unrecoverable media defects are handled by remapping logical block numbers to non-defective locations, with data often being lost when defects “grow” during operation. In MEMS-based storage, most media defects are expected to affect the data under a small number of tips (e.g., 1–4). Therefore, the horizontal ECC can usually be used to reconstruct unavailable bits. This capability is particularly important because the higher density of MEMS-based storage causes a given defect to affect more bits than it would in a disk. Tolerance of large media defects can be further extended by spreading each logical block’s data and ECC among physically distant tips—graph coloring schemes excel at the types of data mappings required.

Tip failures. Failure of a conventional disk’s read/write head or control logic generally renders the entire device inoperable. MEMS-based storage replicates these functions across thousands of components. With so many components, failure of one or more is not only possible, but probable—individual probe tips can break off or “crash” into the media, and fabrication variances will produce faulty tips or faulty tip-specific logic. Most such problems can be handled using the same mechanisms that handle media failures, since failure of a tip or its associated control logic translates into unavail-

ability of data in the corresponding tip region. The horizontal ECC can be used to reconstruct this data.

As with disk drives, spare space needs to be withheld from the fault-free mapping of data to physical locations in MEMS-based storage. This spare space is used to store data that cannot be stored at its default physical location because of media or tip failures. The parallel operation of tips within a track provides an opportunity to avoid the performance and predictability penalties normally associated with defect remapping in disk drives. Specifically, by setting aside one or more *spare tips* in each track, unreadable sectors can be remapped to the same sector under a spare tip. A sector remapped in this way would be accessed at exactly the same time as the original (unavailable) sector would have been. In contrast, disks “slip” LBNs over defective sectors or re-map them to spare sectors elsewhere in a cylinder or zone, changing their access times relative to their original locations.

6.2 Device failures

MEMS-based storage devices are susceptible to similar non-recoverable failures as disk drives: strong external mechanical or electrostatic forces can damage the actuator comb fingers or snap off the springs, manufacturing defects can surface, or the device electronics or channel can fail. These failures should appear and be handled in the same manner as for disks. For example, appropriate mechanisms for dealing with device failures include inter-device redundancy and periodic backups.

Interestingly, MEMS-based storage’s mechanical characteristics are a better match than those of disks for the common read-modify-write operations used in some fault-tolerant schemes (e.g., RAID-5). Whereas conventional disks suffer a full rotation to return to the same sector, MEMS-based storage devices can quickly reverse direction, significantly reducing the read-modify-write latency (Table 2). For the Random workload, a five-disk RAID-5 system has 77% longer response times than a four-disk striping-only system (14.3ms vs. 8.04ms); the latency increase for MEMS-based storage devices is only 27% (1.36ms vs. 1.07ms).

6.3 Recovery from host system crashes

File systems and databases must maintain internal consistency among persistent objects stored on MEMS-based storage devices, just as they do for objects on disks. Although synchronous writes will still hurt performance, the low service times of MEMS-based storage devices will lessen the penalty.

	Atlas 10K		MEMS	
# sectors	8	334	8	334
read	0.14	6.00	0.13	2.19
reposition	5.86	0.00	0.07	0.07
write	0.14	6.00	0.13	2.19
total (ms)	6.14	12.00	0.33	4.45

Table 2: *A comparison of read-modify-write times for 4 KB (8 sector) and disk track-length (334 sector) transfers. Conventional disks must wait for a complete platter rotation during read-modify-write operations, whereas MEMS-based storage devices need only perform a turnaround, a relatively inexpensive operation. This characteristic is particularly helpful for code-based redundancy schemes (for example, RAID-5) or for verify-after-write operations.*

Another relevant characteristic of MEMS-based storage devices is rapid device startup. Since no spindle spin-up time is required, startup is almost immediate—estimated at less than 0.5 ms. In contrast, high-end disk drives can take 15–25 seconds before spin-up and initialization is complete [22]. Further, MEMS-based storage devices do not exhibit the power surge inherent in spinning up disk drives, so power spike avoidance techniques (e.g., serializing the spin-up of multiple disk drives) are unnecessary—all devices can be started simultaneously. Combined, these effects could reduce system restart times from minutes to milliseconds.

7 Other considerations

This section discusses additional issues related to our exploration of OS management for MEMS-based storage devices.

Power conservation. Significant effort has gone into reducing a disk drive’s power consumption, such as reducing active power dissipation and introducing numerous power-saving modes for use during idle times [6, 15, 16]. MEMS-based storage devices are much more energy efficient than disks in terms of operational power. Further, the physical characteristics of MEMS-based storage devices enable a simpler power management scheme: a single idle mode that stops the sled and powers down all non-essential electronics. With no rotating parts and little mass, the media sled’s restart time is very small (estimated at under 0.5 ms). This relatively small penalty enables aggressive idle mode use, switching from active to idle as soon as the I/O queue is empty. Detailed energy breakdown and evaluation indicates that our default MEMS-based storage device employing this immediate-idle scheme would dissipate

only 8–22% of the energy used by today’s low-power disk drives [27].

Alternate seek control models. Our device model assumes that seeks are accomplished by a “slew plus settle” approach, which involves maximum acceleration for the first portion of the seek, followed by maximum deceleration to the destination point and speed, followed by a closed loop settling time. With such seek control, the slew time goes up as the square root of the distance and the settling time is constant (to the first order). The alternate seek control approach, a linear system seek, would incorporate rate proportional feedback to provide damping and a step input force to initiate movement to a desired location and velocity. Seeks based on such a control system exhibit longer seek times (including the settling times) that are much more dependent on seek distance [35]. This should not change our high-level conclusions, but will tend to increase the importance of both SPTF scheduling and subregion data layouts.

Erase cycles. Although our target MEMS-based storage device employs traditional rewriteable magnetic media, some designs utilize media that must be reset before it can be overwritten. For example, the IBM Millipede [31] uses a probe technology based on atomic force microscopes (AFMs), which stores data by melting minute pits in a thin polymer layer. A prominent characteristic of the Millipede design is a block erase cycle requiring several seconds to complete. Such block erase requirements would necessitate management schemes, like those used for Flash RAM devices [5], to hide erase cycle delays.

8 Summary

This paper compares and contrasts MEMS-based storage devices with disk drives and provides a foundation for focused OS management of these new devices. We describe and evaluate approaches for tuning request scheduling, data placement and failure management techniques to the physical characteristics of MEMS-based storage.

One of the general themes of our results is that OS management of MEMS-based storage devices can be similar to, and simpler than, management of disks. For example, disk scheduling algorithms can be adapted to MEMS-based storage devices in a fairly straightforward manner. Also, performance is much less dependent on such optimizations as careful data placement, which can yield order of magnitude improvements for disk-based systems [9, 19, 24]; data placement still matters, but sub-optimal solutions

may not be cause for alarm. In the context of availability, internal redundancy can mask most problems, eliminating both data loss and performance loss consequences common to disk drives. Similarly, rapid restart times allow power-conservation software to rely on crude estimates of idle time.

We continue to explore the use of MEMS-based storage devices in computer systems, including their roles in the memory hierarchy [27] and in enabling new applications.

Acknowledgments

We thank Rick Carley, David Petrou, Andy Klosterman, John Wilkes, the CMU MEMS Laboratory, and the anonymous reviewers for helping us refine this paper. We thank the members and companies of the Parallel Data Consortium (including CLARiON, EMC, HP, Hitachi, Infineon, Intel, LSI Logic, MTI, Novell, PANASAS, Procom, Quantum, Seagate, Sun, Veritas, and 3Com) for their interest, insights, and support. We also thank IBM Corporation and Intel Corporation for supporting our research efforts. John Griffin is supported in part by a National Science Foundation Graduate Fellowship.

References

- [1] L. Richard Carley, James A. Bain, Gary K. Fedder, David W. Greve, David F. Guillou, Michael S. C. Lu, Tamal Mukherjee, Suresh Santhanam, Leon Abelman, and Seungook Min. Single-chip computers with microelectromechanical systems-based magnetic memory. *Journal of Applied Physics*, **87**(9):6680–6685, 1 May 2000.
- [2] Center for Highly Integrated Information Processing and Storage Systems, Carnegie Mellon University. <http://www.ece.cmu.edu/research/chips/>.
- [3] Data Storage Systems Center, Carnegie Mellon University. <http://www.ece.cmu.edu/research/dssc/>.
- [4] Peter J. Denning. Effects of scheduling on file memory operations. *AFIPS Spring Joint Computer Conference* (Atlantic City, New Jersey, 18–20 April 1967), pages 9–21, April 1967.
- [5] Fred Douglass, Ramon Caceres, Frans Kaashoek, Kai Li, Brian Marsh, and Joshua A. Tauber. Storage alternatives for mobile computers. *Symposium on Operating Systems Design and Implementation* (Monterey, CA), pages 25–39. USENIX Association, 14–17 November 1994.
- [6] Fred Douglass, P. Krishnan, and Brian Marsh. Thwarting the power-hungry disk. *Winter USENIX Technical Conference* (San Francisco, CA), pages 292–306. USENIX Association, Berkeley, CA, 17–21 January 1994.
- [7] G. K. Fedder, S. Santhanam, M. L. Reed, S. C. Eagle, D. F. Guillou, M. S.-C. Lu, and L. R. Carley. Laminated high-aspect-ratio microstructures in a conventional CMOS process. *IEEE Micro Electro Mechanical Systems Workshop* (San Diego, CA), pages 13–18, 11–15 February 1996.
- [8] Greg Ganger and Jiri Schindler. Database of validated disk parameters for DiskSim. <http://www.ece.cmu.edu/~ganger/disksim/diskspecs.html>.
- [9] Gregory R. Ganger and M. Frans Kaashoek. Embedded inodes and explicit grouping: exploiting disk bandwidth for small files. *Annual USENIX Technical Conference* (Anaheim, CA), pages 1–17, January 1997.
- [10] Gregory R. Ganger, Bruce L. Worthington, and Yale N. Patt. *The DiskSim Simulation Environment Version 1.0 Reference Manual*, CSE-TR-358-98. Department of Computer Science and Engineering, University of Michigan, February 1998.
- [11] John Linwood Griffin, Steven W. Schlosser, Gregory R. Ganger, and David F. Nagle. Modeling and performance of MEMS-based storage devices. *ACM SIGMETRICS 2000* (Santa Clara, CA, 17–21 June 2000). Published as *Performance Evaluation Review*, **28**(1):56–65, June 2000.
- [12] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*, 2nd ed. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1995.
- [13] Hewlett-Packard Laboratories Storage Systems Program. <http://www.hpl.hp.com/research/storage.html>.
- [14] David M. Jacobson and John Wilkes. *Disk scheduling algorithms based on rotational position*. HPL-CSP-91-7. Hewlett-Packard Laboratories, Palo Alto, CA, 24 February 1991, revised 1 March 1991.
- [15] Kester Li, Roger Kumpf, Paul Horton, and Thomas E. Anderson. A quantitative analysis of disk drive power management in portable computers. *Winter USENIX Technical Conference* (San Francisco, CA), pages 279–291. USENIX Association, Berkeley, CA, 17–21 January 1994.
- [16] Yung-Hsiang Lu, Tajana Šimunić, and Giovanni De Micheli. Software controlled power management. *7th International Workshop on Hardware/Software Codesign* (Rome, Italy), pages 157–161. ACM Press, 3–5 May 1999.
- [17] Marc Madou. *Fundamentals of Microfabrication*. CRC Press LLC, Boca Raton, Florida, 1997.
- [18] Marshall K. McKusick, William N. Joy, Samuel J. Lefler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, **2**(3):181–197, August 1984.
- [19] L. W. McVoy and S. R. Kleiman. Extent-like performance from a UNIX file system. *Winter USENIX Technical Conference* (Dallas, TX), pages 33–43, 21–25 January 1991.
- [20] Microelectromechanical Systems Laboratory, Carnegie Mellon University. <http://www.ece.cmu.edu/~mems/>.
- [21] David A. Patterson, Peter Chen, Garth Gibson, and Randy H. Katz. Introduction to redundant arrays of inexpensive disks (RAID). *IEEE Spring COMPCON* (San Francisco, CA), pages 112–117, March 1989.
- [22] Quantum Corporation. *Quantum Atlas 10K 9.1/18.2/36.4 GB Ultra 160/m SCSI Hard Disk Drive Product Manual*, Publication number 81-119313-05, August 6, 1999.
- [23] Erik Riedel, Christos Faloutsos, Gregory R. Ganger, and David F. Nagle. Data mining on an OLTP system (nearly) for free. *ACM SIGMOD Conference 2000* (Dallas, TX), pages 13–21, 14–19 May 2000.
- [24] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, **10**(1):26–52,

February 1992.

- [25] Chris Ruemmler and John Wilkes. UNIX disk access patterns. *Winter USENIX Technical Conference* (San Diego, CA), pages 405–420, 25–29 January 1993.
- [26] Chris Ruemmler and John Wilkes. *Disk Shuffling*. Technical report HPL-91-156. Hewlett-Packard Company, Palo Alto, CA, October 1991.
- [27] Steven W. Schlosser, John Linwood Griffin, David F. Nagle, and Gregory R. Ganger. Designing computer systems with MEMS-based storage. *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, Massachusetts), 13–15 November 2000. To appear.
- [28] P. H. Seaman, R. A. Lind, and T. L. Wilson. On teleprocessing system design, part IV: an analysis of auxiliary-storage activity. *IBM Systems Journal*, 5(3):158–170, 1966.
- [29] Margo Seltzer, Peter Chen, and John Ousterhout. Disk scheduling revisited. *Winter USENIX Technical Conference* (Washington, DC), pages 313–323, 22–26 January 1990.
- [30] T. J. Teorey and T. B. Pinkerton. A comparative analysis of disk scheduling policies. *Communications of the ACM*, 15(3):177–184, March 1972.
- [31] P. Vettiger, M. Despont, U. Drechsler, U. Dürig, W. Häberle, M. I. Lutwyche, H. E. Rothuizen, R. Stutz, R. Widmer, and G. K. Binnig. The “Millipede”—more than one thousand tips for future AFM data storage. *IBM Journal of Research and Development*, 44(3):323–340, 2000.
- [32] Paul Vongsathorn and Scott D. Carson. A system for adaptive disk rearrangement. *Software—Practice and Experience*, 20(3):225–242, March 1990.
- [33] Kensall D. Wise. Special issue on integrated sensors, microactuators, and microsystems (MEMS). *Proceedings of the IEEE*, 86(8):1531–1787, August 1998.
- [34] Bruce L. Worthington, Gregory R. Ganger, and Yale N. Patt. *Scheduling for modern disk drives and non-random workloads*. CSE-TR-194-94. Department of Computer Science and Engineering, University of Michigan, March 1994.
- [35] Pu Yang. *Modeling probe-based data storage devices*. Technical report. Department of Computer Science, University of California Santa Cruz, June 2000. Master’s thesis.

Trading Capacity for Performance in a Disk Array*

Xiang Yu[†] Benjamin Gum[†] Yuqun Chen[†]

Randolph Y. Wang[†] Kai Li[†] Arvind Krishnamurthy[‡] Thomas E. Anderson[§]

Abstract

A variety of performance-enhancing techniques, such as striping, mirroring, and rotational data replication, exist in the disk array literature. Given a fixed budget of disks, one must intelligently choose what combination of these techniques to employ. In this paper, we present a way of designing disk arrays that can flexibly and systematically reduce seek and rotational delay in a balanced manner. We give analytical models that can guide an array designer towards optimal configurations by considering both disk and workload characteristics. We have implemented a prototype disk array that incorporates the configuration models. In the process, we have also developed a robust disk head position prediction mechanism without any hardware support. The resulting prototype demonstrates the effectiveness of the configuration models.

1 Introduction

In this paper, we set out to answer a simple question: how do we systematically increase the performance of a disk array by adding more disks?

This question is motivated by two phenomena. The first is the presence of a wide variety of performance-enhancing techniques in the disk array literature. These include striping[17], mirroring[3], and replication of data within a track to improve rotational delay[18]. All of these techniques share the common theme of improving performance by scaling the number of disks. Their performance impacts, however, are different. Given a fixed budget of disks, an array designer faces the choice of what combination of these techniques to use.

The second phenomenon is the increasing cost and performance gap between disk and memory. This increase is fueled by the explosive growth of disk areal density, which is at an annual rate of about 60% [8]. On the other hand, the areal density of memory has been improving at a rate of only 40% per year [8]. The result is a cost gap of roughly two orders of magnitude today.

As disk latency has been improving at about only 10% per year [8], disks are becoming increasingly unbalanced in terms of the relationship between capacity and latency. Although cost per byte and capacity per drive remain the predominant concerns of a large sector of the market, a substantial performance-sensitive (and, in particular, latency-sensitive) market exists. Database vendors today have already recognized the importance of building a balanced secondary storage system. For example, in order to achieve high performance on TPC-C [26], vendors configure systems based on the number of disk heads instead of capacity. To achieve D times the bandwidth, the heads form a D -way mirror, a D -way stripe, or a RAID-10 configuration [4, 11, 25], which combines mirroring and striping so that each unit of the striped data is also mirrored. What is not well understood is how to configure the heads to get the most out of them.

The key contributions of this paper are:

- a flexible strategy for configuring disk arrays and its performance models,
- a software-only disk head position prediction mechanism that enables a range of position-sensitive scheduling algorithms, and
- evaluation of a range of alternative strategies that trade capacity for performance.

More specifically, we present a disk array configuration, called an SR-Array, that flexibly combines striping with rotational replication to reduce both seek and rotational delay. The power of this configuration lies in that it can be flexibly adjusted in a balanced manner that takes a variety of parameters into consideration. We present a series of analytical models that show how to configure the array by considering both disk and workload characteristics.

*This work was supported in part by the Scalable I/O project under the DARPA grant DABT63-94-C-0049 and by the National Science Foundation under grant CDA-9624099 and CCR-9984790.

[†]Department of Computer Science, Princeton University, {xyu,gum,yuqun,rywang,li}@cs.princeton.edu.

[‡]Department of Computer Science, Yale University, arvind@cs.yale.edu.

[§]Department of Computer Science and Engineering, University of Washington, Seattle, tom@cs.washington.edu.

To evaluate the effectiveness of this approach, we have designed and implemented a prototype disk array that incorporates the SR-Array configurations. In the process, we have developed a method for predicting the disk head location. It works on a wide range of off-the-shelf hard drives without special hardware support. This mechanism is not only a crucial ingredient in the success of the SR-Array configurations, it also enables the implementation of rotational position sensitive scheduling algorithms, such as Shortest Access Time First (SATF) [14, 23], across the disk array. Because these algorithms involve inter-disk replicas, without the head-tracking mechanism, it would have been difficult to choose replicas intelligently even if the drives themselves perform sophisticated internal scheduling.

Our experimental results demonstrate that the SR-Array provides an effective way of trading capacity for improved performance. For example, under one file system workload, a properly configured six-disk SR-Array delivers 1.23 to 1.42 times lower latency than that achieved on highly optimized striping and mirroring systems. The same SR-Array achieves 1.3 to 2.6 times better sustainable throughput while maintaining a 15 ms response time on this workload.

The remainder of the paper is organized as follows. Section 2 presents the SR-Array analytical models that guide configuration of disk arrays. Section 3 describes the integrated simulator and prototype disk array that implement the SR-Array configuration models. Section 4 details the experimental results. Section 5 describes some of the related work. Section 6 concludes.

2 Techniques and Analytical Models

In this section, we provide a systematic analysis of how a combination of the performance-enhancing techniques such as striping and data replication can contribute to seek distance reduction, rotational delay reduction, overall latency improvement, and throughput improvement. These analytical models, though approximations in some cases, serve as a basis for configuring a disk array for a given workload.

2.1 Reducing Seek Distance

We start by defining the following abstract problem: suppose the maximum seek distance on a single disk is S , the total amount of data fits on a single disk, and accesses are uniformly distributed across the data set. Then, how can we effectively employ D disks to reduce the average seek latency? We use seek distance to simplify our presentation. (Seek la-

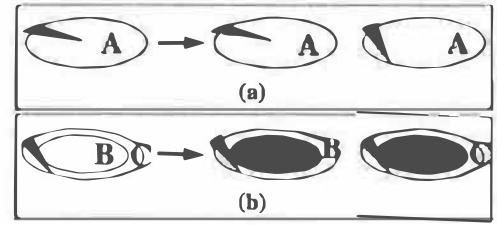


Figure 1: Techniques for reducing seek distance. Capital letters represent a portion of the data. To the left of the arrows, we show how data is (logically) stored on a single disk. To the right, we show different ways that data on the single disk can be distributed on two disks ($D = 2$): (a) D -way mirroring, and (b) D -way striping.

tency is approximately a linear function of seek distance only for long seeks [22].) As a base case, one can show that the average seek distance for reads on a single disk [24] is $S_1 = S/3$.

The first seek reduction technique is D -way mirroring (shown in Figure 1(a)). D -way mirroring can reduce seek distance because we can choose the disk head that is closest to the target sector in terms of seek distance. With D disks, the average seek distance is the average of the minimum of D random variables [3], which is $S/(2D + 1)$.

The second technique is striping (and keeping disks partially empty). Figure 1(b) illustrates a two-way striping. Data on the original single disk is partitioned into two disjoint sets: B and C . We store B on the outer edge of the first disk and C on the outer edge of the second disk. The space in the middle of these two disks is not used. In this case, the single large disk is in effect split into two smaller disks. As a result, the disk head movement is restricted to a smaller region. Assuming constant track capacity and uniform accesses, Matloff [17] gives the average seek distance of a D -way stripe (S_s):

$$S_s(D) = \frac{S}{3D} \quad (1)$$

The amount of seek reduction achieved by striping is better than that of D -way mirroring. However, D -way mirroring provides reliability through the use of multiple copies. A hybrid scheme would provide reliability along with smaller seek latencies. RAID-10, widely used in practice, is a concrete example of such a hybrid scheme: in a RAID-10 system, data is striped across D_s disks while each block is also replicated on D_m different disks.

2.2 Reducing Rotational Delay

As we reduce the average seek distance, the rotational delay starts to dominate the disk access cost. To address this limitation, we replicate data at different rotational positions, and by choosing a replica

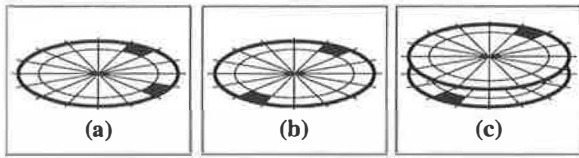


Figure 2: Techniques for reducing rotational delay. (a) Randomly placed replicas. (b) Evenly spaced replicas. (c) Replicas placed on different tracks (either within a single disk or on different disks).

that is rotationally closest to the disk head, we can reduce rotational delay. Replication for reducing rotational delay can increase seek distance by pushing data farther apart. We will discuss combining the techniques for reducing seek and rotation distance in a later section.

If the time needed to complete a rotation on a single disk is R , we observe that the average rotational delay $R_r(1)$ is simply half of a full rotation, i.e. $R_r(1) = R/2$. If we replicate data D times, and spread the replicas evenly on a track (i.e. $360/D$ degrees apart from each other as shown in Figure 2(b)), the average read rotational latency R_r is:

$$R_r(D) = \frac{R}{2D} \quad (2)$$

We can also show that the average read rotational latency is $R_r = R/(D+1)$, if we randomly place replicas (shown in Figure 2(a)) on the same track. This technique is therefore less beneficial than evenly distributing the replicas and is not used in our design.

However, having multiple replicas on one track increases average rotational latency R_w for writing all these replicas to:

$$R_w(D) = R - \frac{R}{2D} \quad (3)$$

Of course, we could reduce the write costs by writing the closest copy synchronously and propagating other copies during idle periods. Equation (3) gives the worst case cost when we are not able to mask the replica propagation. Notice that $R_r(D) + R_w(D) = R$. Thus if reads are more frequent than writes, making more replicas will reduce overall latency. If reads and writes are equally frequent, varying D will not change the average overall latency. If writes are more frequent than reads, the approach with no replication is always the best. Note that this relationship is independent of the value of R and is only true for foreground replica propagation. Background propagation may make replication desirable even when writes outnumber reads.

Figures 2(a) and (b) illustrate the concept of rotational replication by making copies within the same track. Unfortunately, this decreases the bandwidth of large I/O as a result of shortening the effective track length and increasing track switch frequency. To avoid unnecessary track switches, we place the replicas on different tracks either within a cylinder of a single disk or on different disks (shown in Figure 2(c)). Track skews must be re-arranged so that large sequential I/Os that cross track boundaries do not suffer any unnecessary degradation.

2.3 Reducing Both Seek and Rotational Delay

In the previous sections, we have discussed existing techniques for reducing seek distance and rotational delay in isolation. Their combined effects, however, are not well understood. We now develop models that predict the overall latency as we increase the number of disks.

SR-Array: Combining Striping and Rotational Replication

Since disk striping reduces seek distance and rotational replication reduces rotational delay, we can combine the two techniques to reduce overall latency. We call the resulting configuration an *SR-Array*. Figure 3 shows an example SR-Array. In an SR-Array, we perform rotational replication on the same disk. We explore rotational replication on different disks in a later section.

Given a fixed budget of D disks, we would now like to answer the following question: what degree of striping and what degree of rotational replication should we use for the best resulting performance? We call this the “aspect ratio” question. We first consider this question for random access latency, and then we examine how the model can be extended to take into account other workload parameters.

Read Latency on an SR-Array

In this paper, we define *overhead* to include various processing times, transfer costs, track switch time, and mechanical acceleration/deceleration times. We focus on the overhead-independent part of the latency in the following analysis.

Let us assume that we have a single disk’s worth of data, and we have a total of D disks. Suppose the maximum seek time on one disk is S , the time for a full rotation is R , only $1/D_s$ of the cylinders on a single disk is used to limit the seek distance, and D_r is the number of replicas for reducing rotational delay ($D_s D_r = D$). If $D_r = 1$, an SR-Array degenerates to simple striping and only $1/D$ of the

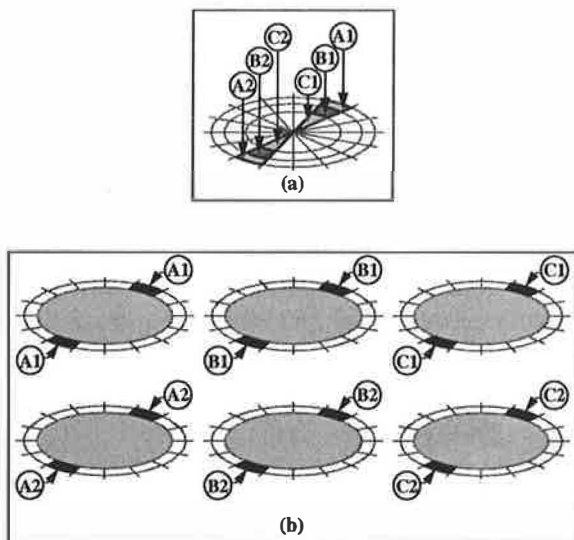


Figure 3: Reducing both seek and rotational delay in an SR-Array. (a) Data on an original disk is divided into six parts. (b) A 3×2 SR-Array. Each disk holds only one sixth of the original data. The two rotational replicas for each block ensure that the maximum rotational delay for any data is half of a full rotation. (Two times the number of disks are needed to support two-way rotational replication; this is shown in the vertical dimension.) The rotational replicas expand the seek distance between different data blocks so the maximum seek distance on each of these six disks is the same as that in a simple three-way striped system (denoted by the three disks in the horizontal dimension).

available space is used. If $D_s = 1$, we use all the available space. In Figure 3, $D_s = 3$ and $D_r = 2$.

Because the random read latency is the sum of the overhead, the average seek time, and the average rotational time, we can approximate the overhead-independent part of random read latency $T_R(D_s, D_r)$ as:

$$T_R(D_s, D_r) = \frac{S}{3D_s} + \frac{R}{2D_r} \quad (4)$$

Given the constraint of $D_s D_r = D$, we can prove that the following configuration produces the best overall latency for independent random I/Os under low load:

$$\begin{cases} D_s = \sqrt{\frac{2S}{3R}D} \\ D_r = \sqrt{\frac{3R}{2S}D} \end{cases} \quad (5)$$

The overhead-independent part of latency under this configuration is therefore:

$$T_{best} = \sqrt{\frac{2SR}{3D}} \quad (6)$$

It is likely that the optimal D_s and D_r are not integer values. For such scenarios, we choose D_r to be

the maximum integer factor of D that is less than or equal to the optimal non-integer value.

Disks with slow rotational speed (large R) demand a higher degree of rotational replication. In terms of the SR-Array illustration of Figure 3(b), this argues for a tall thin grid. Conversely, disks with poor seek characteristics (large S) demand a large striping factor. In terms of Figure 3(b), this argues for a short fat grid. The model indicates that the latency improvement on an SR-Array is proportional to the square root of the number of disks (\sqrt{D}).

So far, our discussion of the model applies to random access by assuming an average seek of $S/3$ in Equation (6). To capture seek locality, we replace $S/3$ with the average seek of a workload. In the later experimental sections, this is accomplished by dividing $S/3$ with a “seek locality index” (L), which is observed from the workload. The model does not directly account for sequential access.

Read/Write Latency on an SR-Array

Now we extend the latency model of an SR-Array to model the performance of both read and write operations. When performing a write, in the worst case scenario of not being able to mask the cost of replica propagation, we must incur a write latency of $T_W(D_s, D_r)$:

$$T_W(D_s, D_r) = \frac{S}{3D_s} + R - \frac{R}{2D_r} \quad (7)$$

Let the number of reads be X_r , the number of writes that can be propagated in the background be X_{wb} , and the number of writes that are propagated in the foreground be X_{wf} . We define the ratio p :

$$p = \frac{X_r + X_{wb}}{X_r + X_{wb} + X_{wf}} \quad (8)$$

The average read/write latency, $T(D_s, D_r) = pT_R + (1 - p)T_W$, can be expressed as:

$$T(D_s, D_r) = \frac{S}{3D_s} + p\frac{R}{2D_r} + (1 - p)(R - \frac{R}{2D_r}) \quad (9)$$

The first term is the average seek incurred by any request. The second term is the average rotational delay consumed by I/O operations that do not result in foreground replica propagation (based on Equation (2)) with probability p ; and the third term is the rotational delay consumed by writes whose replicas are propagated in the foreground due to lack of idle time (based on Equation (3)) with probability $1 - p$. We can prove that the following configuration provides the best overall latency:

$$\begin{cases} D_s = \sqrt{\frac{2S}{3R(2p-1)}} D \\ D_r = \sqrt{\frac{3R(2p-1)}{2S}} D \end{cases} \quad (10)$$

The latency under this configuration is:

$$T_{best} = \sqrt{\frac{2SR(2p-1)}{3D}} + (1-p)R \quad (11)$$

A low p ratio calls for a short fat grid in Figure 3(b). A p ratio under 50% precludes rotational replication and pure striping provides the best configuration. In the best case, when all write replicas can be propagated in the background (or when we have no writes at all), writes and reads become indistinguishable as far as this model is concerned, so p approaches 1 and the latency improvement is proportional to \sqrt{D} .

2.4 Scheduling and Throughput

We now consider throughput improvements and address the following questions: 1) How do we schedule the requests to take advantage of the additional resources? 2) How do we modify the SR-Array aspect ratio models to optimize for throughput?

Scheduling on an SR-Array

In our SR-Array design, we choose to place a block and all its replicas (if any) on a single disk. Requests are sent to the only disk responsible for the data, which queues requests and performs scheduling on each disk locally and independently. In contrast, in a mirrored system, because any request can be scheduled for any copy, devising a good global scheduler is non-trivial. We report heuristics-based results for mirrored systems in later sections. In this section, we focus on scheduling for an SR-Array and develop an extension of the LOOK algorithm for an SR-Array, which we call RLOOK.

Under the traditional LOOK algorithm, the disk head moves bi-directionally from one end of the disk to another, servicing requests that can be satisfied by the cylinder under the head. On an SR-Array disk, in addition to scanning the disk like LOOK in the seek direction, our RLOOK scheduling also chooses the replica that is rotationally closest among all the replicas during the scan.

Suppose q is the number of requests to be scheduled for a single RLOOK stroke on a single disk, and S , R , D_s , D_r , D , and p retain their former definitions from Section 2.3, the average time of a single request in the stroke is $T(D_s, D_r)$:

$$T(D_s, D_r) = \frac{S}{qD_s} + p\frac{R}{2D_r} + (1-p)(R - \frac{R}{2D_r}) \quad (12)$$

The first term amortizes q requests over the end-to-end seek time, which is an approximation of the time needed for a LOOK stroke. The two remaining terms are identical to those of Equation (9). (Empirically, this is a good approximation when $q > 3$. When $q \leq 3$, the requests are so sparse that the latency models of Equations (9) through (11) are used instead.)

Starting with Equation (12), we can prove that the best latency is achieved with the following configuration:

$$\begin{cases} D_s = \sqrt{\frac{2S}{R(2p-1)q}} D \\ D_r = \sqrt{\frac{R(2p-1)q}{2S}} D \end{cases} \quad (13)$$

Under this configuration, the average request latency of RLOOK is:

$$T_{best} = \sqrt{\frac{2SR(2p-1)}{qD}} + (1-p)R \quad (14)$$

Assuming that each request has an overhead of T_o , we can approximate the single disk throughput by

$$N_1 = \frac{1}{T_o + T_{best}} \quad (15)$$

In addition to the parameters that we have seen in the previous models, the aspect ratio is now also sensitive to q , a measure of the busyness of the system. A long queue allows for the amortization of the end-to-end seek over many requests; consequently, we should devote more disks to reducing rotational delay. In terms of the SR-Array illustration of Figure 3(b), this argues for a tall thin grid. As with the model in the last section, a p ratio under 50% also precludes rotational replication; pure striping is best and Equations (13) through (15) do not apply. In the best case, when all replicas are propagated in the background, p approaches 1, and the model suggests that the overhead-independent part of service time also improves proportionally to \sqrt{D} .

Having modeled the throughput of a single disk, we attempt to model the throughput of an SR-Array with D disks and a total of $Q = Dq$ outstanding requests, where q is the average queue size per disk. We assume that the requests are randomly distributed in the system. There could be a load imbalance in the form of idle disks when Q is not much more than D . The probability of one disk being idle is $(1 - \frac{1}{D})^Q$. Therefore, the total throughput of the system is:

$$N_D \approx D \left[1 - \left(1 - \frac{1}{D} \right)^Q \right] \cdot N_1 \quad (16)$$

Although this approximation is derived based on reasoning about the presence of idle disks, we shall see in Section 4.2 that it is in fact a good approximation of more general cases.

Now that we have described the RLOOK extension to LOOK, it is easy to understand a similar extension to SATF: RSATF. An RSATF scheduler chooses the next request with the shortest access time by considering all rotational replicas. It is well known that SATF outperforms LOOK [14, 23] by considering rotational delay. Our experimental results will show that the gap between RLOOK and RSATF is smaller because both scheduling algorithms consider rotational delays. Once the detailed low level disk layout is understood, RLOOK is simple to implement; it is an attractive local scheduler for an SR-Array.

2.5 Comparing SR-Array with Striped Mirror

In an SR-Array, all replicas exist on the same disk. Removing this restriction, we can place these replicas at rotationally even positions on different disks in a “synchronized” mirror, a mirrored system whose spindles are synchronized. We call this layout strategy a *striped mirror*, one flavor of the RAID-10 systems known in the disk array industry. (RAID-10 is a broader term that typically does not necessarily imply the requirement of synchronized spindles and the placement of replicas at rotationally even positions.) To make the performance of the striped mirror competitive to a corresponding SR-Array, we must choose replicas intelligently based on rotational positioning information.

Even with these assumptions, a striped mirror is not equivalent to an SR-Array counterpart. Consider a simple example involving only two disks: blocks A and B reside on different disks in an SR-Array; but each of the disks in a corresponding striped mirror has both blocks. Now consider a reference stream of AAB. On an SR-Array, the two accesses to A are satisfied by two rotational replicas on one disk, consuming less than a full rotation in terms of rotational delay; and the access to B is satisfied by a different disk so its access time is independent of the first disk and the first two accesses. In an attempt to emulate the behavior of the SR-Array, we must send the two accesses to A to the two replicas on different disks in a striped mirror; but now the access time of B is affected by the first two accesses because both disks are busy. In general, it is impossible to enforce identical individual access time for a stream of requests to an SR-Array and a striped mirror.

Statistically, the read latency of a striped mirror should be slightly better than the latency on a corresponding SR-Array. This is because the average of the minimum of the sum of seek and rotational delay is smaller than the sum of the average seek and average minimum rotational delay.

In terms of throughput, the simple example above shows that for an arbitrary stream of requests, there does not exist a general schedule on a striped mirror that is equivalent to that on a corresponding SR-Array. We do not pretend to know how to optimally choose replicas on a striped mirror. Section 3 discusses a number of heuristics. The performance of our best effort implementation of a striped mirror has failed to match that of an SR-Array counterpart.

In terms of feasibility, as spindle synchronization is becoming a rarer feature on modern drives, one can only approximate striped mirrors on unsynchronized spindles. In terms of reliability, a striped mirror is obviously better than an SR-Array.

In general, it is possible to combine an SR-Array with a striped mirror to achieve the benefits of both approaches so that some of the replicas are on the same disk and some are on different ones. The result is the most general configuration: a $D_s \times D_r \times D_m$ “SR-Mirror”, where D_s implies that only $1/D_s$ of the space is used (to reduce seek time), D_r is the number of replicas on the same disk, and D_m is the number of replicas on different disks. A $D \times 1 \times 1$ system is D -way striping. A $1 \times 1 \times D$ system is a D -way mirror. A $D_s \times D_r \times 1$ system is an SR-Array. A $D_s \times 1 \times 2$ is the most common RAID-10 configuration. We may approximate the performance of an SR-Mirror by replacing D_r in the SR-Array models with $D_r \times D_m$.

2.6 Summary of Techniques and Models

In this section, we have explored how by scaling the number of disks in a storage system we can 1) reduce seek distance, 2) reduce rotational delay, 3) reduce overall latency by combining these techniques in a balanced manner, and 4) improve throughput. To achieve these goals, the storage system needs to be configured based on a number of parameters. We have developed simple models that capture the following parameters that influence the configuration decisions:

- disk characteristics in the form of seek and rotational characteristics (S and R),
- read/write ratio (p),
- busyness of the system (q), and
- seek locality (L).

We note that there does not exist a single “perfect” SR-Array configuration; instead, there may exist one “right” configuration for every workload and cost/performance specification. As we increase the number of disks, and if we properly configure the storage system, under the right conditions, the various models of this section suggest the following rule of thumb: *By using D disks, we can improve the overhead-independent part of response time by a factor of \sqrt{D} .*

3 Implementation

In the previous section, we analyzed how to configure a disk array to deliver better performance as we scale the number of disks in the system. In this section, we describe the prototype MimdRAID implementation that puts the theory to test.

3.1 Overview

To show the feasibility of our approach, we have developed an infrastructure for experimenting with the various techniques. Figure 4 illustrates the system components and how they stack against each other. There are a number of ways of configuring the system. The controlling agent (at the top) can be either a user level disk driver or an OS kernel device driver. The devices (at the bottom) can be either a disk simulator or real SCSI disks. The remaining components (in the middle) are shared across all configurations and are built in a layered fashion.

The *SCSI Abstraction Layer* abstracts device specific operations such as SCSI device detection at boot time, issuing SCSI commands, and retrieval of command status. We currently support two different 10000 RPM SCSI drives: Seagate ST34502LW and ST39133LWV; but all experimental results reported are based on the ST39133LWV.

The *Calibration Layer* is used for calibrating the disk and extracting information regarding the physical layout of the disk. It keeps track of where the disk head is currently located and calculates how much time is required to move the head from its current position to a target sector. Section 3.2 provides more details about our head-tracking technique.

A parallel layer to the SCSI abstraction layer is the *Simulator*. We decided to integrate the simulator into the architecture to shorten the simulation time for long traces: we not only eliminate idle time, but also replace I/O time (which can be long) with simulated time. The simulator also provides the flexibility of exploring the impact of changing disk characteristics. To faithfully simulate the behavior of the disks that we currently use in the prototype, the

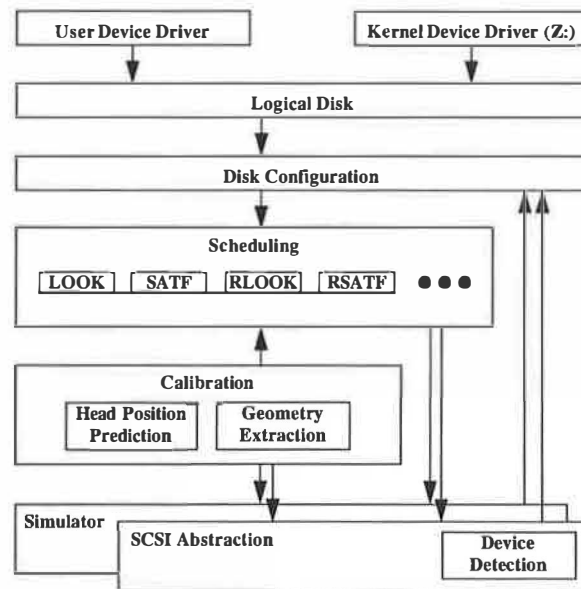


Figure 4: Prototype architecture.

simulator receives timing information from the calibration layer to configure itself.

The *Scheduling layer* implements several disk scheduling policies. This layer maintains a read/write command queue for each physical disk and invokes a user determined policy to pick the next request at each scheduling step. We call these queues the *drive queues*. Sections 3.3 and 3.4 provide details of the scheduling policies.

The *Disk Configuration Layer* provides support for configuring a collection of disks using techniques such as D -way mirroring, striping, SR-Array, and SR-Mirror. It translates I/O requests for a logical disk to a set of I/O commands on the physical disks and inserts them into the appropriate drive queues. The striping unit is 64K bytes in our experiments.

The topmost *Logical Disk Layer* is in charge of exposing the logical disk to the application. The kernel device driver exposes a mount point (e.g. drive letter Z: in Windows 2000) to user space. The user level driver exposes the disk in the form of an API to the application.

3.2 Predicting Disk Head Position

The techniques presented in Section 2 rely on the driver’s ability to accurately predict the disk head location and the cost of disk operations such as track switches, seeks, and rotational placement. The driver also needs information on the layout of the physical sectors on the disk.

Previous proposals that depend on the knowledge of head positions have relied on hardware support [5, 27]. Unfortunately, this level of support is not always available on commodity drives. We have

developed a software-only head-tracking method. Our scheme requires issuing read accesses to a fixed *reference sector* at periodic intervals. The head tracking algorithm computes the disk head position based on the time stamp taken immediately after the completion of the most recent read operation of the reference sector. The basic idea is that the time between two read accesses of the reference sector is always an integral multiple of the full rotation time plus an unpredictable OS and SCSI overhead. By gradually increasing the time interval between adjacent read requests to the reference sector, we amortize the overhead of reading the reference sector. Our experiments show that periodic re-calibration at an interval of two minutes yields predictions that have an error of only 1% of a full rotation time with 98% confidence. It is encouraging that we can achieve a high degree of accuracy with a low overhead associated with reading the reference sector every two minutes. To further reduce this overhead, we can exploit the timing information and known disk head location at the end of a request. We have not implemented this optimization.

To obtain an accurate image of the disk, we use methods that are similar to those used by Worthington [29] for determining the logical to physical sector mapping. We obtain information on disk zones, track skew, bad sectors, and reserved sectors through a sequence of low-level disk operations.

The last piece of information that we measure is the cost of performing track switches and seeks. Small errors in these timing measurements may introduce a penalty that is close to a full rotation. To reduce the number of rotation misses, we introduce a slack of k sectors so that when the mechanism predicts the head to be less than k sectors away from the target, the scheduler conservatively chooses the next rotational replica after the target. This slack can be adjusted by a real time feedback loop to ensure that more than 99% of the requests are on target.

3.3 Scheduling Reads

The head-tracking mechanism, along with accurate models of the disk layout and the seek profile, allows us to implement sophisticated local scheduling policies on individual disks; these include RLOOK, SATF, and RSATF.

Scheduling on a mirrored system, however, is more complex due to the fact that a request can be serviced by any one of the disks that have the data. We use the following heuristic scheduling algorithm. When a read request arrives, if some of the disks that contain the data are idle, the scheduler immediately sends the request to the idle disk head that is

closest to a copy of the data. If all disks that contain the desired data are busy, the logical disk layer duplicates the request and inserts the copies into the drive queues of all these disks. As soon as such a request is scheduled on one disk, all other duplicate requests are removed from all other drive queues. When a disk completes processing a request, its local scheduler greedily chooses the “nearest” request from its own drive queue. Although this heuristic algorithm may not be optimal, it can avoid load imbalance and works fairly well in practice.

3.4 Delayed Writes

While multiple copies of data reduce read latency, they present a challenge for performing writes efficiently because more than one copy needs to be written. We need to make $D_r \times D_m$ copies for a $D_s \times D_r \times D_m$ SR-Mirror. It is however feasible to propagate the copies lazily when the disks are idle. We can issue a write to the closest copy and delay writing the remaining copies. For back-to-back writes to the same data block, which happens frequently for data that die young [21], we can safely discard unfinished updates from previous writes.

In our implementation, we maintain for each disk a *delayed write queue*, which is distinct from the foreground request queue. When a write request arrives, we initially schedule the first write using the foreground request queue just as we do for reads. As soon as writing one of the replicas is scheduled, we set aside the remaining update operations for the other replicas in the individual delayed write queues. Entries from this queue are serviced when the foreground request queue becomes empty. Delayed writes require us to make a copy of the data because the original buffer is returned to the OS as soon as the first write completes.

To provide crash recovery, the physical location of the first write is stored in a *delayed write metadata table* that is kept in NVRAM. Note that it is not necessary to store a copy of the data itself in NVRAM—the physical location of the first write is sufficient for completing the remaining delayed writes upon recovery; so the table is small. When the metadata table fills up to a threshold (10,000 entries in the current implementation), we force delayed writes out by moving them to the foreground request queue.

3.5 Validating the Integrated Simulator

So far, we have described the architecture and the components of the integrated MimdRAID simulator and device driver. To establish 1) the accuracy of the head-tracking mechanism, and 2) the validity of

Operating system	Microsoft Windows 2000
CPU type	Intel Pentium III 733 MHz
Memory	128 MB
SCSI Interface	Adaptec 39160
SCSI bus speed	160 MB/s
Disk model	Seagate ST39133LWV 9.1 GB
RPM	10000
Average seek	5.2 ms read, 6.0 ms write

Table 1: Platform characteristics.

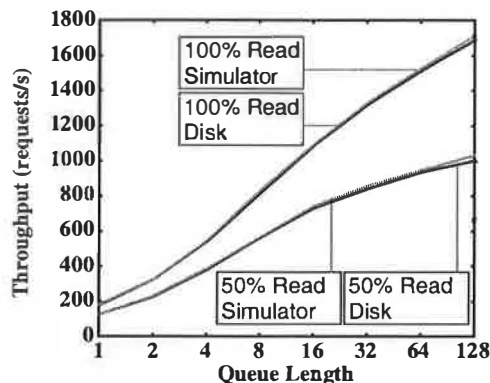


Figure 5: Comparison of throughput results from the prototype system and the simulator. We use two random workloads, one with just reads, and another with an equal number of reads and writes. The request size is 512 bytes. The array configuration is a 2×3 SR-Array based on the RSATF scheduler. Writes are synchronously propagated in the foreground. We vary the number of outstanding requests (on the x-axis).

the simulator, we perform a series of experiments using “Iometer”, a benchmark developed by the Intel Server Architecture Lab [13]. Iometer can generate different workloads of various characteristics including read/write ratio, request size, and the maximum number of outstanding requests. We use Iometer to generate equivalent workloads to drive both the device driver and the simulator. Table 1 lists some platform characteristics of the prototype. Figure 5 shows the Iometer result. The throughput discrepancy between the simulator and the prototype under all queueing conditions is under 3%.

To shed more light on the accuracy of the model, in Table 2, we give more detailed statistics of subjecting the model and the prototype to the “Cello base” file system workload (described in Section 4.1). The mean prediction error and low standard deviation show that there are essentially only two types of requests: 99.8% of the predictions are almost right on target, and 0.2% of the predictions miss their targets by a very small amount of time and incur a full rotation penalty. The net effect of these rare rotation misses, however, is insignificant in terms of overall access time. These results indicate that the

Misses	0.22%
Mean Prediction Error	3 μ s
Standard Deviation of Error	31 μ s
Average Access Time	2746 μ s
Demerit	52 μ s
Demerit/Access Time	1.9 %

Table 2: Detailed statistics of model accuracy when subjected to the “Cello base” file system workload. The configuration is a 2×3 SR-Array based on RSATF scheduling. I/O requests in this experiment are physical I/O requests sent to drives; and access time is that of a physical I/O. Prediction error is the difference between the access time predicted by the scheduler and the actual measured access time of a single request. We calculate demerit using the definition by Ruemmler and Wilkes [21].

simulator faithfully simulates a real SR-Array, allowing us to understand the behavior of the SR-Array using simulation-based results in later sections.

4 Experimental Results

In this section, we evaluate the performance of the prototype MimdRAID under two sets of experiments. The first set of experiments is based on playing real-world file system and transaction processing traces on the MimdRAID simulator. The second set of experiments is based on running on the prototype itself a synthetic workload generator that is designed to stress it in ways beyond what is possible with the traces. The purposes of the experiments are to: 1) validate the models of Section 2, 2) show the effectiveness and importance of workload-driven configuration, and 3) demonstrate the use of scaling the number of disks as a cost-effective means of improving performance for certain workloads.

4.1 Macro-benchmarks

We test our system using two traces. Cello is a two month trace taken on a server running at HP Labs [21]. It had eight disks and was used for running simulations, compilations, editing, and reading mail and news. We use one week’s worth of trace data (for the period of 5/30/92 through 6/6/92). TPC-C is a disk trace (collected on 5/03/94) of an unaudited run of the Client/Server TPC-C benchmark running at approximately 1150 tpmC on a 100 Warehouse database.

Logical Data Sets

The 9.1 GB Seagate disks that we use are much larger and faster than any of the original disks used in the trace; therefore, we do not map the original small disks one-to-one onto our large disks. Instead, we group the original data into three logical data

	Cello base	Cello disk 6	TPC-C
Data size	8.4 GB	1.3 GB	9.0 GB
I/Os	1,717,483	1,545,341	3,598,422
Duration	1 week	1 week	2 hours
Avg. I/O rate	2.84/s	2.56/s	500/s
Reads	55.2%	35.8%	54.8%
Async. writes	18.9%	16.1%	0
Seek locality (L)	4.14	16.67	1.04
Read after write (1 hour)	4.15%	3.8%	14.8%

Table 3: Trace characteristics. The “seek locality” row is calculated as the ratio between the average of random seek distances on that disk and the average seek distance observed in the trace. This ratio is used to adjust the S parameter when applying the models of Section 2 in subsequent discussions. The “read after write (1 hour)” row lists the percentage of I/O operations that are reads that occur less than one hour after writing the same data.

sets and study how to place each data set in a disk array made of new disks.

The first data set consists of all the Cello disk data with the exception of disk 6, which houses “/usr/spool/news”. We merge these separate Cello disk traces based on time stamps to form a single large trace. The data from different disks are concatenated to form a single data set. We refer to this workload as “Cello base” in the rest of the paper. The second data set consists solely of Cello disk 6. This disk houses the news directory; it exhibits access patterns that are different from the rest of the Cello disks and accounts for 47% of the total accesses. We refer to this workload as “Cello disk 6” in the rest of the paper. The third data set consists of data from 31 original disks of the TPC-C trace. We merge these traces to form a single large trace; and we concatenate these disks to form a single data set as well. We refer to this workload as “TPC-C”.

Table 3 lists the key characteristics of the trace data sets. Of particular interest is the last row, which reports the fraction of I/Os that are reads to recently written data. Although this ratio is high for TPC-C, it does not rise higher for intervals longer than an hour. Together with the amount of available idle time, this ratio impacts the effectiveness of the delayed write propagation strategy and influences the array configurations.

To test our system with various load conditions, we also uniformly scale the rate at which the trace is played based on the time stamp information. For example, when the scaling rate is two, the traced inter-arrival times are halved.

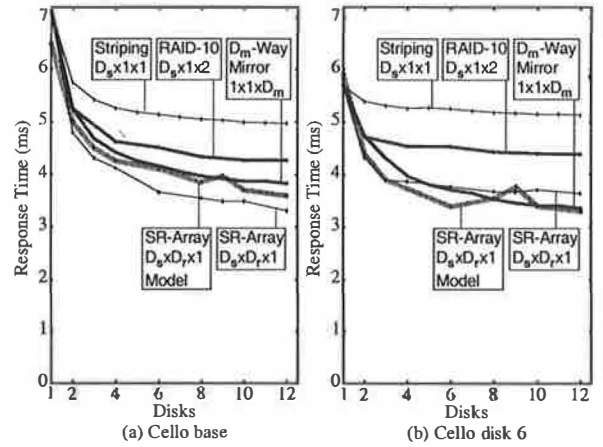


Figure 6: Comparison of average I/O response time of the Cello file system workloads on different disk array configurations. The SR-Array uses the RSATF scheduler and the remaining configurations use the SATF scheduler. The two configurations labeled as “RAID-10” and “D_m-way mirror” are reliable configurations and are denoted by thicker curves. This convention is used throughout the rest of the figures.

Playing Cello Traces at Original Speed

As a starting point, we place the Cello base data set on one Seagate disk and the Cello disk 6 data set on another. Although we have fewer number of spindles in this case than in the original trace, the original speed of the Cello traces is still sufficiently low that we are effectively measuring individual I/O latency most of the time. There is also sufficient idle time to mask the delayed write propagations. Therefore, we apply the model in Section 2.3 (Equation (5)) to configure the SR-Array. When applying the formulas, we account for the different degree of seek locality (L) in Table 3 by replacing S with S/L . We perform replica propagation in the background for all configurations. Although all write operations from the trace are played, we exclude those of asynchronous writes when reporting response time; most of the asynchronous writes are generated by the file system sync daemon at 30 second intervals. All reported response times include an overhead of 2.7 ms, which includes various processing times, transfer costs, track switch time, and mechanical acceleration/deceleration times, as defined in Section 2.3.

Figure 6 shows the performance improvement on the Cello workloads as we scale the number of disks under various configurations. The curve labeled as “SR-Array” shows the performance of the best SR-Array configuration for a given number of disks. The SR-Array performs well because it is able to effectively distribute disks to the seek and rotational dimensions in a balanced manner. In contrast, the performance of simple striping is poor due to the

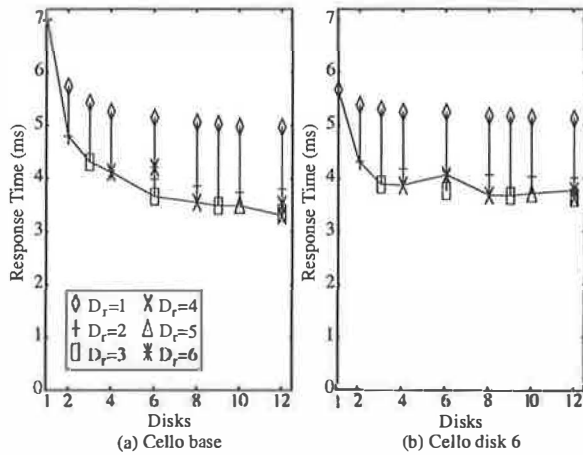


Figure 7: Configurations of the SR-Array for the two workloads of Figure 6. The curves show the performance of the SR-Array configuration recommended by the model of Equation (5). Each point symbol in the graph shows the performance of an alternative SR-Array configuration with a different number of rotational replicas (D_r).

lack of rotational delay reduction. This effect is more apparent for larger numbers of disks due to the diminishing returns from seek distance reduction. The performance of RAID-10 is intermediate because the two replicas allow for a reduction in the rotational delay to a limited extent. D -way mirroring is the closest competitor to an SR-Array because of its high degree of flexibility in choosing which replica to read. (We will expose the weakness of D -way mirroring in subsequent experiments.) Note that our SATF-based implementation of RAID-10 and D -way mirroring are highly optimized versions based on rotational positioning knowledge.

The figure also shows that the latency model of Section 2.3 is a good approximation of the SR-Array performance. The anomalies on the model curves are due to the two following practical constraints: 1) D_s and D_r must be integer factors of the total number of disks D , and 2) our implementation restricts the largest degree of rotational replication to six. This restriction is due to the difficulty of propagating more copies within a single rotation, as rotational replicas are placed on different tracks and a track switch costs about $900 \mu s$. Due to these constraints, for example, the largest practical value of D_r for $D = 9$ is only three, much smaller than the non-integer solution of Equation (5) (5.8 for Cello base and 11.6 for Cello disk 6).

While the Cello base data set consumes an entire Seagate disk, the Cello disk 6 data set only occupies about 15% of the space on a single Seagate disk; so the maximum seek delay of Cello disk 6 is small to begin with for all configurations. Consequently, a larger D_r for an SR-Array is desirable as we in-

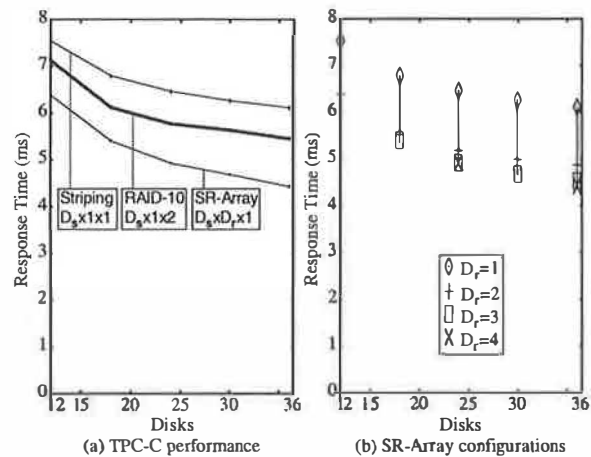


Figure 8: Average I/O response time of the TPC-C trace. (a) Comparison of striping, RAID-10, and SR-Array. (b) Comparison of alternative configurations of an SR-Array.

crease the number of disks. With these large D_r values, however, the practical constraints enumerated above start to take effect. Coupled with the fact that seek time is no longer a linear function of seek distance at such short seek distances, this explains the slightly more pronounced anomalies of the SR-Array performance with a large number of disks on the Cello disk 6 workload.

Figure 7 compares the performance of other possible SR-Array configurations with that of the configuration chosen by the model. For example, when the number of disks is six, the model recommends a configuration of $D_s \times D_r = 2 \times 3$ for Cello base. The three alternative configurations are 1×6 , 3×2 , and 6×1 . The figure shows that the model is largely successful at finding good SR-Array configurations. For example, on Cello base, with six disks, the SR-Array is 1.23 times as fast as a highly optimized RAID-10, 1.42 times as fast as a striped system, and 1.94 times as fast as the single disk base case.

Playing the TPC-C Trace at Original Speed

Although a single new Seagate disk can accommodate the entire TPC-C data set in terms of capacity, it cannot support the data rate of the original trace. Indeed, only a fraction of the space on the original traced disks was used to boost the data rate. We start with 12 disks for each of the array configurations. Figure 8 shows the performance as we scale the number of disks beyond the starting point. The data rate experienced by each disk under this workload is much higher than that under the Cello system described in the last section. The workload also contains a large fraction of writes so it also stresses delayed write propagation as idle periods are shorter.

Compared to Figure 6, two curves are missing

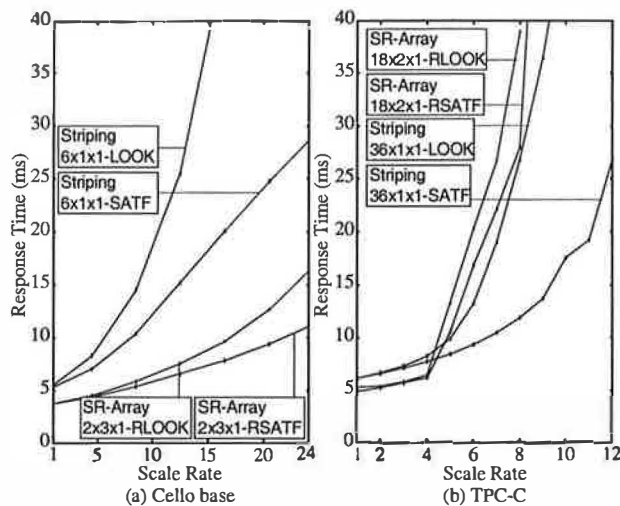


Figure 9: Comparison of local disk schedulers for different configurations as we raise the I/O rate. We use six disks for Cello base (a), and 36 for TPC-C (b).

from Figure 8. One is D -way mirroring—it is impossible to support the original data rate while attempting to propagate D replicas for each write. Another missing curve is the latency model—the high data rate renders the latency model inaccurate. The spirit of Figure 8, however, is very much similar to that of Figures 6 and 7: a properly configured SR-Array is faster than a RAID-10, which is faster than a striped system.

What is more interesting is the fact that striping, the only configuration that does not involve replication, is not the best configuration even under the high update rate exhibited by this workload. There are at least two reasons. First, even under this higher I/O rate, there are still idle periods to mask replica propagations. Second, even without idle periods, there exists a tradeoff between the benefits received from reading the closest replicas and the cost incurred when propagating replicas, as demonstrated by the models of Section 2.2; a configuration that can successfully exploit this tradeoff excels. For example, with 36 disks, a $9 \times 4 \times 1$ SR-Array is 1.23 times as fast as a $18 \times 1 \times 2$ RAID-10, and 1.39 times as fast as a $36 \times 1 \times 1$ striped system.

Playing Traces at Accelerated Rate

Although the original I/O rate of TPC-C is higher than that of the Cello traces, it does not stress the 12-disk arrays discussed in the last section. We now raise the I/O rates to stress the various configurations. For example, when the “scale rate” is two, we halve the inter-arrival time of requests.

Before we compare the different array configurations, we first consider the impact of the local

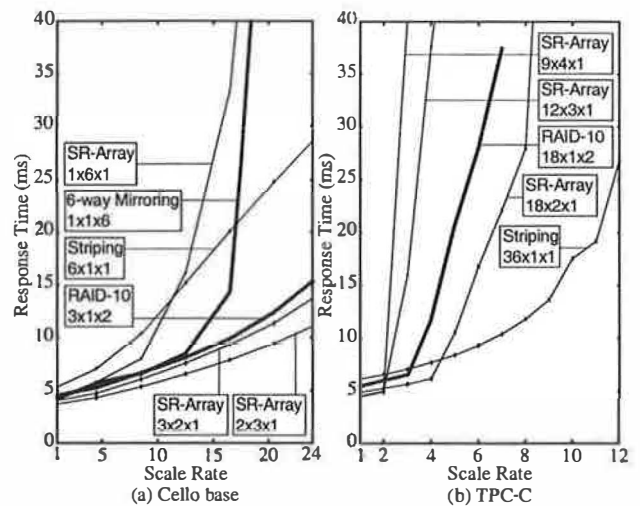


Figure 10: Comparison of I/O response time on different configurations as we raise the I/O rate. We use six disks for Cello base (a), and 36 for TPC-C (b).

disk schedulers. Figure 9 evaluates four schedulers: LOOK and SATF for striping, and RLOOK and RSATF for an SR-Array. Given a particular request arrival rate, the gap between RLOOK and RSATF is smaller than that between LOOK and SATF. This is because both RLOOK and RSATF take rotational positioning into consideration. Although it is a well known result that SATF out-performs LOOK, we see that SATF alone is not sufficient for addressing rotational delays if the array is mis-configured to begin with. For example, under the Cello base workload, a $2 \times 3 \times 1$ SR-Array significantly out performs a $6 \times 1 \times 1$ striped system even if the former only uses an RLOOK scheduler while the latter uses an SATF scheduler. In the rest of the discussions, unless specified otherwise, we use the RSATF scheduler for SR-Arrays and the SATF scheduler for other configurations.

Figure 10 shows the performance of the various configurations while we fix the number of disks for each workload and vary the rate at which the trace is played. Under the Cello base workload (shown in Figure 10(a)), the 6-way mirror and the 1×6 SR-Array deliver the lowest sustainable rates. These configurations make the largest number of replicas and it is difficult to mask the replica propagation under high request rates. 6-way mirroring is better than the 1×6 SR-Array, because the 6-way mirror can afford the flexibility of choosing any disk to service a request, so it can perform better load balancing. The 2×3 SR-Array is best for all the arrival rates that we have examined; this is because the benefits derived from the extra rotational replicas outweigh the cost. If we demand an average response time no greater than 15 ms, the $2 \times 3 \times 1$ SR-Array

can support a request rate that is 1.3 times that of a $3 \times 1 \times 2$ RAID-10 and 2.6 times that of a $6 \times 1 \times 1$ striped system.

The situation is different for the TPC-C workload (shown in Figure 10(b)). Under the original trace playing rate, the $9 \times 4 \times 1$ SR-Array is best. As we raise the request arrival rate, we must successively reduce the degree of replication; so the role of the best configuration passes to the $12 \times 3 \times 1$, $18 \times 1 \times 2$, $18 \times 2 \times 1$, and finally, $36 \times 1 \times 1$ configurations, in that order. If we again demand an average response time no greater than 15 ms, the $36 \times 1 \times 1$ configuration can support a request rate that is 1.3 times that of a $18 \times 2 \times 1$ configuration and 2.1 times that of a $18 \times 1 \times 2$ RAID-10 configuration.

Comparison Against Memory Caching

We have seen that it is possible to achieve significant performance improvement by scaling the number of disks. We now compare this approach against one alternative: simply adding a bigger volatile memory cache. The memory cache performs LRU replacement. Synchronous writes are forced to disks in both alternatives. In the following discussion, we assume that the price per MB ratio between memory and disk is M . At the time of this writing, 256 MB of memory costs \$300, an 18 GB SCSI disk costs \$400, and these prices give an M value of 57.

Figure 11(a) examines the impact of memory caching on the Cello base workload. At the trace scale rate of one, we need to cache an additional 1.5%, or 126 MB, of the file system in memory to achieve the same performance improvement of doubling the number of disks; and we need to cache 4%, or 336 MB, of the file system to reach the performance of a four-disk SR-Array. M needs to be less than 67 and 75 respectively in order for memory caching to be cost effective, which it is today.

At the trace scale rate of three, using similar reasoning, we can conclude that M needs to be less than 20 in order for memory caching to be more cost effective than doubling the number of disks. Beyond this budget, at this I/O rate, the diminishing locality and the need to flush writes to disks make the addition of memory less attractive. The addition of disks, however, speeds up all I/O operations, albeit at a diminishing rate.

Figure 11(b) examines the impact of memory caching on the TPC-C workload, which has much less locality. We start with a 12-disk SR-Array. At a scale rate of one, M needs to be less than 80 in order for memory caching to be a cost effective alternative to increasing the number of disks to 18 or 24. Adding memory is a more attractive alternative.

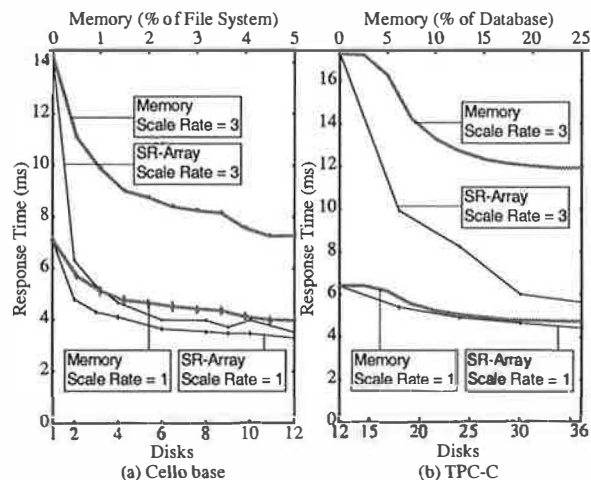


Figure 11: Comparison of the effects of memory caching and scaling the number of disks. The two SR-Array curves show the performance improvement achieved by scaling the number of disks (bottom x-axis) and they correspond to playing the traces at the original speed and three times the original speed. The two Memory curves show the performance improvement achieved by scaling the amount of memory caching (top x-axis).

At a scale rate of three, M needs to be less than 24 for memory caching to be more cost effective than increasing the number of disks to 18. Beyond this budget, adding memory provides little additional performance gain, while increasing the number of disks from 18 to 36 can provide an additional 1.76 times speedup.

4.2 Micro-benchmarks

To further explore the behavior of the prototype, we use the Intel Iometer benchmark to stress some array configurations in a controlled manner. In all the following micro-benchmarks, we use a seek locality index of 3, as defined in Section 2.3. We measure the throughput in these experiments.

Throughput Models

In this experiment, we perform only random read operations on the disk array while maintaining a constant number of outstanding requests. (We examine writes more fully in the next subsection.) The goals are 1) to understand the scalability of the disk array, 2) to understand the behavior of the system under different load conditions, and 3) to validate (part of) the throughput model of Section 2.4.

Figure 12 shows that the SR-Array using the RSATF scheduler scales well as we increase the number of disks under this Iometer workload. The RLOOK scheduler is a close approximation of the RSATF scheduler; and the RLOOK-based throughput model closely captures the behavior of the SR-

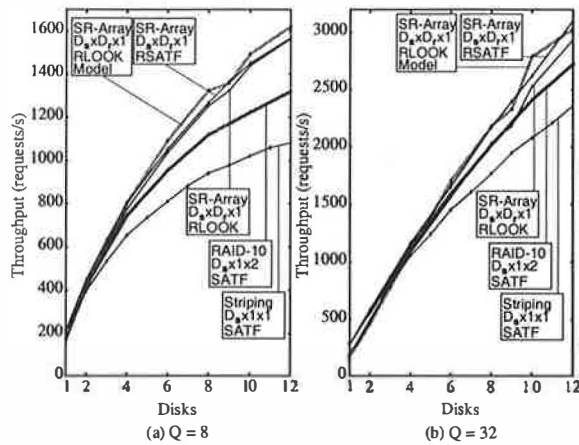


Figure 12: Throughput as a function of array configuration, number of disks, and queue length. The queue lengths are (a) 8, and (b) 32.

Array, including the throughput degradation experienced when the queue length is short.

The SATF-based striped and RAID-10 systems do not scale as well as the SR-Array. The throughput gap between all these systems, however, narrows as the queue length increases since the SATF scheduler can overcome the lack of rotational replicas when it has a large number of requests to choose from.

Replica Propagation Cost

We now analyze configurations under *foreground* write propagation and validate the model in Section 2.4.

Figure 13 shows the throughput results as Iometer maintains a constant queue length of mixed reads and writes. Each write leads to immediate replica propagations; so write ratio and foreground write ratio are the same, namely, $1 - p$, where p is defined by Equation (8) of Section 2.3.

Among the configurations shown in the figure, RAID-10 has the worst performance under high write ratios. To understand why, consider the propagation of a single write: the $3 \times 2 \times 1$ SR-Array requires a single seek followed by writing 2 rotational replicas in a single cylinder; but a corresponding $3 \times 1 \times 2$ RAID-10 requires 2 seeks so the amount of arm movement tends to be greater.

The performance of the striped $6 \times 1 \times 1$ configurations degrade slightly for high write ratios as writes are slightly more expensive than reads.

The performance difference between a $3 \times 2 \times 1$ SR-Array and a $6 \times 1 \times 1$ striped system depends on the write ratio with the former better for low write ratios. If we only consider rotational delay, the rotational replication model of Section 2.2 would

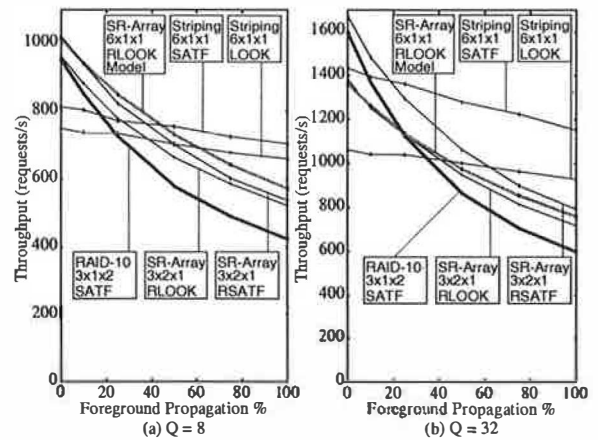


Figure 13: Throughput as a function of foreground write propagation rate and queue length. The total number of disks is six. The queue lengths are (a) 8, and (b) 32.

imply that the cross-over point between them under LOOK/RLOOK scheduling should be close to the 50% write ratio. If we also consider seek distance, the $3 \times 2 \times 1$ SR-Array has worse seek performance so the actual cross-over point is less than 50%.

Because SATF benefits a $6 \times 1 \times 1$ configuration more than RSATF does to a $3 \times 2 \times 1$ configuration, the cross-over point between these systems under SATF/RSATF scheduling is to the left of that under LOOK/RLOOK scheduling. This distance is even greater when the queue is longer (in Figure 13(b)).

The figure also shows that the RLOOK throughput model (Equation (16)) closely tracks the experimental result under varying write ratios.

5 Related Work

A number of previous storage systems were designed to take into consideration the tradeoff between capacity and performance. Hou and Patt [10] performed a simulation study of the tradeoff between mirroring and RAID-5.

The HP AutoRAID incorporated both mirroring and RAID-5 into a two-level hierarchy [28]. The mirrored upper level provided faster small writes at the expense of consuming more storage, while the RAID-5 lower level was more frugal in its use of disk space. Its primary focus was solving the small write problem of RAID-5.

We have taken the tradeoff between capacity and performance a step further by 1) improving latency and throughput of all I/O operations, 2) being able to benefit from more than twice the excess capacity, and 3) providing a means of systematically configuring the extra disk heads.

The HP Ivy project [15] was a simulation study of how a high degree of replication could improve read

performance. Our study differs from Ivy in several ways. First, Ivy only explored reducing seek distance and left rotational delay unresolved. Second, Ivy only examined mirroring. The third difference is a feature of Ivy that we intend to incorporate into our system in the future: Ivy dynamically chose the candidate and the degree of replication by observing access patterns. We are currently researching a wide range of access patterns (including those at the file system level) that can be used to dynamically tune the array configuration.

Matloff [17] derived a model of linear improvement of seek distance as one increased the number of disks devoted to striping. Bitton and Gray derived a model of seek distance reduction [3] and studied seek scheduling [2] for a D -way mirror. Neither study considered the impact of rotational delay.

Dishon and Liu [6] considered latency reduction on either synchronized or unsynchronized D -way mirrors. A synchronized mirror can reduce foreground propagation latency because the multiple copies can be written at nearly the same time if we insist that the replicas are placed at rotationally identical positions. This advantage comes at the cost of poor read latency because it allows no rotational delay reduction for reads.

Polyzois [20] proposed careful scheduling of delayed writes to different disks in a mirror to maximize throughput, a technique that can potentially benefit delayed writes in our systems when the replicas are on different disks.

The “distorted mirror” [19] provided an alternative way of improving the performance of writes in a mirror. It performed writes initially to rotationally optimal but variable locations and propagated them to fixed locations later. This technique can be integrated with our delayed write strategy as well.

Lumb et al. [16] exploited “free bandwidth” that is available when the disk head is in between servicing normal requests in a busy system. The free bandwidth was used for background I/O activity. Propagating replicas in our system is a good use of this free bandwidth.

Ng examined intra-track replication as a means of reducing rotational delay [18]. We extend this approach to improve large I/O bandwidth by performing rotational replication across different tracks.

The importance of reducing rotational delay has long been recognized. Seltzer and Jacobson independently examined a number of disk scheduling algorithms that take rotational position into consideration [14, 23]. Our work considers the impact of reducing rotational delay in array configurations in a manner that balances the conflicting goal of reduc-

ing seek and rotational delay at the same time.

At the time of this writing, the Trail system [12] independently developed a disk head tracking mechanism that is similar to ours. Trail used this information to perform fast log writes to carefully chosen rotational positions. A similar write strategy was in use in the earlier Mime system [5], but Mime relied on hardware support for its rotational positioning information. Aboutabl et al. developed a similar disk timing measurement strategy, which was used to model the response time of individual I/O requests [1].

A number of drive manufacturers have incorporated SATF-like scheduling algorithms in their firmware. An early example was the HP C2490A [9]. Our host-based software solution enables the employment of such scheduling on drives that do not support it internally. Furthermore, it allows experimentation with strategies such as rotational replica selection, strategies that would have been difficult to realize even on drives that support intelligent scheduling internally. On the other hand, if the drive does support intelligent internal scheduling, an interesting question that this study has not addressed is how we can adapt our algorithm for such drives without relying on complex predictions.

One of our goals of studying the impact of altering array configurations is to understand how to configure a storage system given certain cost/performance specifications. The “attribute-managed storage” project [7] at HP shares this goal, although its focus is at the disk array level as opposed to individual drive level.

6 Conclusion

In this paper, we have described a way of designing disk arrays that can flexibly reduce seek and rotational delay in a balanced manner. We have presented a series of analytical models that take into consideration disk and workload characteristics. By incorporating these models and a robust software-based disk head position prediction mechanism, the MimdRAID prototype can deliver latency and throughput results unmatched by conventional approaches.

Acknowledgement

We would like to thank Doug Clark, Ed Grochowski, Ed Lee, Spencer Ng, Chandu Thekkath, Honesty Young, and the class participants of Princeton CS598e (spring 2000) for the early discussions on this topic, HP Labs for supplying the I/O traces,

the OSDI reviewers for their comments, and John Wilkes for a large number of excellent suggestions during a meticulous and tireless shepherding process.

References

- [1] ABOUTABL, M., AGRAWALA, A., AND DECOTIGNIE, J.-D. Temporally Determinate Disk Access: An Experimental Approach (Extended Abstract). In *Proc. of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Madison, Wisconsin, June 1998), pp. 280-281.
- [2] BITTON, D. Arm Scheduling in Shadowed Disks. In *Proc. of 34th IEEE COMPCON* (San Francisco, CA, February 1989), pp. 132-136.
- [3] BITTON, D., AND GRAY, J. Disk Shadowing. In *Proc. of the Fourteenth International Conference on Very Large Data Bases* (Los Angeles, CA, August 1988), Morgan Kaufmann, pp. 331-338.
- [4] BORR, A. Transaction Monitoring in Encompass: Reliable Distributed Transaction Processing. In *Proc. of the Seventh International Conference on Very Large Data Bases* (Cannes, France, September 1981), IEEE Press, pp. 155-165.
- [5] CHAO, C., ENGLISH, R., JACOBSON, D., STEPANOV, A., AND WILKES, J. Mime: a High Performance Parallel Storage Device with Strong Recovery Guarantees. Tech. Rep. HPL-CSP-92-9 rev 1, Hewlett-Packard Company, Palo Alto, CA, March 1992.
- [6] DISHON, Y., AND LUI, T. S. Disk Dual Copy Methods and Their Performance. In *Proc. of Eighteenth International Symposium on Fault-Tolerant Computing (FTCS-18)* (Tokyo, Japan, 1988), IEEE CS Press, pp. 314-318.
- [7] GOLDING, R., SHRIVER, E., SULLIVAN, T., AND WILKES, J. Attribute-managed Storage. In *Workshop on Modeling and Specification of I/O* (San Antonio, TX, October 1995).
- [8] GROWCHOWSKI, E. Emerging Trends in Data Storage on Magnetic Hard Disk Drives. In *Datatech* (September 1988), ICG Publishing, pp. 11-16.
- [9] HEWLETT-PACKARD COMPANY, PALO ALTO, CA. *HP C2490A 3.5-inch SCSI-2 Disk Drives Technical Reference Manual (HP Part No. 5961-4359)*, 3rd edition. Boise, Idaho, 1993.
- [10] HOU, R., AND PATT, Y. N. Trading Disk Capacity for Performance. In *Proc. of the Second International Symposium on High Performance Distributed Computing* (Spokane, WA, July 1993), pp. 263-270.
- [11] HSIAO, H.-I., AND DEWITT, D. J. Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines. In *Proc. of the 1990 IEEE International Conference on Data Engineering* (February 1990), pp. 456-465.
- [12] HUANG, L., AND CHIUH, T. Trail: Write Optimized Disk Storage System. <http://www.ecsl.cs.sunysb.edu/trail.html>.
- [13] INTEL SERVER ARCHITECTURE LAB. Iometer: The I/O Performance Analysis Tool for Servers. <http://developer.intel.com/design/servers/devtools/iometer>.
- [14] JACOBSON, D. M., AND WILKES, J. Disk Scheduling Algorithms Based on Rotational Position. Tech. Rep. HPL-CSP-91-7rev1, Hewlett-Packard Company, Palo Alto, CA, February 1991.
- [15] LO, S.-L. Ivy: A Study on Replicating Data for Performance Improvement. Tech. Rep. HPL-CSP-90-48, Hewlett-Packard Company, Palo Alto, CA, December 1990.
- [16] LUMB, C., SCHINDLER, J., GANGER, G. R., RIEDEL, E., AND NAGLE, D. F. Towards Higher Disk Head Utilization: Extracting "Free" Bandwidth from Busy Disk Drives. In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation* (San Diego, CA, October 2000).
- [17] MATLOFF, N. S. A multiple disk system for both fault tolerance and improved performance. *IEEE Transactions on Reliability R-36*, 2 (June 1987), 199-201.
- [18] NG, S. W. Improving disk performance via latency reduction. *IEEE Transactions on Computers* 40, 1 (January 1991), 22-30.
- [19] ORJI, C. U., AND SOLWORTH, J. A. Doubly Distorted Mirrors. In *Proc. of ACM SIGMOD Conference* (May 1993), pp. 307-316.
- [20] POLYZOIS, C., BHIDE, A., AND DIAS, D. Disk Mirroring with Alternating Deferred Updates. In *Proc. of the Nineteenth International Conference on Very Large Data Bases* (Dublin, Ireland, 1993), Morgan Kaufmann, pp. 604-617.
- [21] RUEMLER, C., AND WILKES, J. UNIX Disk Access Patterns. In *Proc. of the Winter 1993 USENIX* (San Diego, CA, Jan. 1993), Usenix Association, pp. 405-420.
- [22] RUEMLER, C., AND WILKES, J. An Introduction to Disk Drive Modeling. *IEEE Computer* 27, 3 (March 1994), 17-28.
- [23] SELTZER, M., CHEN, P., AND OUSTERHOUT, J. Disk Scheduling Revisited. In *Proc. of the 1990 Winter USENIX* (Washington, D.C., Jan. 1990), Usenix Association, pp. 313-323.
- [24] TEOREY, T. J., AND PINKERTON, T. B. A comparative analysis of disk scheduling policies. *Communications of ACM* 15, 3 (March 1972), 177-184.
- [25] TERADATA CORP. *DBC/1012 Database Computer System Manual Release 2.0*, November 1985.
- [26] TRANSACTION PROCESSING PERFORMANCE COUNCIL. *TPC Benchmark C Standard Specification*. Waterside Associates, Fremont, CA, August 1996.
- [27] WANG, R. Y., ANDERSON, T. E., AND PATTERSON, D. A. Virtual Log Based File Systems for a Programmable Disk. In *Proc. of the Third Symposium on Operating Systems Design and Implementation* (New Orleans, LA, February 1999), Operating Systems Review, Special Issue, pp. 29-43.
- [28] WILKES, J., GOLDING, R., STAELIN, C., AND SULLIVAN, T. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems* 14, 1 (February 1996).
- [29] WORTHINGTON, B. L., GANGER, G. R., PATT, Y. N., AND WILKES, J. On-Line Extraction of SCSI Disk Drive Parameters. In *Proc. of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Ottawa, Canada, May 1995), Performance Evaluation Review 23(1), pp. 146-156.

Interposed Request Routing for Scalable Network Storage

Darrell C. Anderson, Jeffrey S. Chase, Amin M. Vahdat *

Department of Computer Science

Duke University

{anderson,chase,vahdat}@cs.duke.edu

Abstract. This paper explores interposed request routing in Slice, a new storage system architecture for high-speed networks incorporating network-attached block storage. Slice interposes a request switching filter — called a *μproxy* — along each client's network path to the storage service (e.g., in a network adapter or switch). The *μproxy* intercepts request traffic and distributes it across a server ensemble. We propose request routing schemes for I/O and file service traffic, and explore their effect on service structure.

The Slice prototype uses a packet filter *μproxy* to virtualize the standard Network File System (NFS) protocol, presenting to NFS clients a unified shared file volume with scalable bandwidth and capacity. Experimental results from the industry-standard SPECsfs97 workload demonstrate that the architecture enables construction of powerful network-attached storage services by aggregating cost-effective components on a switched Gigabit Ethernet LAN.

1 Introduction

Demand for large-scale storage services is growing rapidly. A prominent factor driving this growth is the concentration of storage in data centers hosting Web-based applications that serve large client populations through the Internet. At the same time, storage demands are increasing for scalable computing, multimedia and visualization.

A successful storage system architecture must scale to meet these rapidly growing demands, placing a premium on the costs (including human costs) to administer and upgrade the system. Commercial systems increasingly interconnect storage devices and servers with dedicated Storage Area Net-

works (SANs), e.g., FibreChannel, to enable incremental scaling of bandwidth and capacity by attaching more storage to the network. Recent advances in LAN performance have narrowed the bandwidth gap between SANs and LANs, creating an opportunity to take a similar approach using a general-purpose LAN as the storage backplane. A key challenge is to devise a distributed software layer to unify the decentralized storage resources.

This paper explores *interposed request routing* in Slice, a new architecture for network storage. Slice interposes a request switching filter — called a *μproxy* — along each client's network path to the storage service. The *μproxy* may reside in a programmable switch or network adapter, or in a self-contained module at the client's or server's interface to the network. We show how a simple *μproxy* can virtualize a standard network-attached storage protocol incorporating file services as well as raw device access. The Slice *μproxy* distributes request traffic across a collection of storage and server elements that cooperate to present a uniform view of a shared file volume with scalable bandwidth and capacity.

This paper makes the following contributions:

- It outlines the architecture and its implementation in the Slice prototype, which is based on a *μproxy* implemented as an IP packet filter. We explore the impact on service structure, reconfiguration, and recovery.
- It proposes and evaluates request routing policies within the architecture. In particular, we introduce two policies for transparent scaling of the name space of a unified file volume. These techniques complement simple grouping and striping policies to distribute file access load.
- It evaluates the prototype using synthetic benchmarks including SPECsfs97, an industry-standard workload for network-attached storage servers. The results demonstrate that the

*This work is supported by the National Science Foundation (EIA-9972879 and EIA-9870724), Intel, and Myricom. Anderson is supported by a U.S. Department of Education GAANN fellowship. Chase and Vahdat are supported by NSF CAREER awards (CCR-9624857 and CCR-9984328).

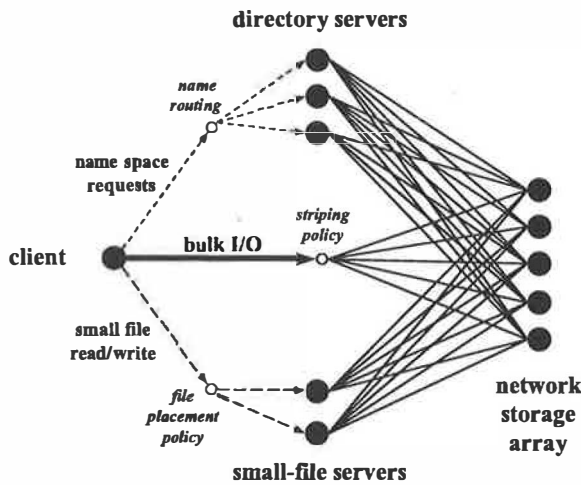


Figure 1: Combining functional decomposition and data decomposition in the Slice architecture.

system is scalable and that it complies with the Network File System (NFS) V3 standard, a popular protocol for network-attached storage.

This paper is organized as follows. Section 2 outlines the architecture and sets Slice in context with related work. Section 3 discusses the role of the μ proxy, defines the request routing policies, and discusses service structure. Section 4 describes the Slice prototype, and Section 5 presents experimental results. Section 6 concludes.

2 Overview

The Slice file service consists of a collection of servers cooperating to serve an arbitrarily large *virtual volume* of files and directories. To a client, the ensemble appears as a single file server at some virtual network address. The μ proxy intercepts and transforms packets to redirect requests and to represent the ensemble as a unified file service.

Figure 1 depicts the structure of a Slice ensemble. Each client's request stream is partitioned into three functional request classes corresponding to the major file workload components: (1) high-volume I/O to large files, (2) I/O on small files, and (3) operations on the name space or file attributes. The μ proxy switches on the request type and arguments to redirect requests to a selected server responsible for handling a given class of requests. Bulk I/O operations route directly to an array of *storage nodes*, which provide block-level access to raw storage objects. Other operations are distributed among specialized file managers responsible for small-file I/O

and/or name space requests.

This *functional decomposition* diverts high-volume data flow to bypass the managers, while allowing specialization of the servers for each workload component, e.g., by tailoring the policies for disk layout, caching and recovery. A single server node could combine the functions of multiple server classes; we separate them to highlight the opportunities to distribute requests across more servers.

The μ proxy selects a target server by switching on the request type and the identity of the target file, name entry, or block, using a separate routing function for each request class. Thus the routing functions induce a *data decomposition* of the volume data across the ensemble, with the side effect of creating or caching data items on the selected managers. Ideally, the request routing scheme spreads the data and request workload in a balanced fashion across all servers. The routing functions may adapt to system conditions, e.g., to use new server sites as they become available. This allows each workload component to scale independently by adding resources to its server class.

2.1 The μ proxy

An overarching goal is to keep the μ proxy simple, small, and fast. The μ proxy may (1) rewrite the source address, destination address, or other fields of request or response packets, (2) maintain a bounded amount of soft state, and (3) initiate or absorb packets to or from the Slice ensemble. The μ proxy does not require any state that is shared across clients, so it may reside on the client host or network interface, or in a network element close to the server ensemble. The μ proxy is not a barrier to scalability because its functions are freely replicable, with the constraint that each client's request stream passes through a single μ proxy.

The μ proxy functions as a network element within the Internet architecture. It is free to discard its state and/or pending packets without compromising correctness. End-to-end protocols (in this case NFS/RPC/UDP or TCP) retransmit packets as necessary to recover from drops in the μ proxy. Although the μ proxy resides "within the network", it acts as an extension of the service. For example, since the μ proxy is a layer-5 protocol component, it must reside (logically) at one end of the connection or the other; it cannot reside in the "middle" of the connection where end-to-end encryption might hide layer-5 protocol fields.

2.2 Network Storage Nodes

A shared array of network storage nodes provides all disk storage used in a Slice ensemble. The μ proxy routes bulk I/O requests directly to the network storage array, without intervention by a file manager. More storage nodes may be added to incrementally scale bandwidth, capacity, and disk arms.

The Slice block storage prototype is loosely based on a proposal in the National Storage Industry Consortium (NSIC) for object-based storage devices (OBSD) [3]. Key elements of the OBSD proposal were in turn inspired by the CMU research on Network Attached Secure Disks (NASD) [8, 9]. Slice storage nodes are “object-based” rather than sector-based, meaning that requesters address data as logical offsets within *storage objects*. A storage object is an ordered sequence of bytes with a unique identifier. The placement policies of the file service are responsible for distributing data among storage objects so as to benefit fully from all of the resources in the network storage array.

A key advantage of OBSDs and NASDs is that they allow for cryptographic protection of storage object identifiers if the network is insecure [9]. This protection allows the μ proxy to reside outside of the server ensemble’s trust boundary. In this case, the damage from a compromised μ proxy is limited to the files and directories that its client(s) had permission to access. However, the Slice request routing architecture is compatible with conventional sector-based storage devices if every μ proxy resides inside the service trust boundary.

This storage architecture is orthogonal to the question of which level arranges redundancy to tolerate disk failures. One alternative is to provide redundancy of disks and other vulnerable components internally to each storage node. A second option is for the file service software to mirror data or maintain parity across the storage nodes. In Slice, the choice to employ extra redundancy across storage nodes may be made on a per-file basis through support for mirrored striping in our prototype’s I/O routing policies. For stronger protection, a Slice configuration could employ redundancy at both levels.

The Slice block service includes a *coordinator* module for files that span multiple storage nodes. The coordinator manages optional block maps (Section 3.1) and preserves atomicity of multisite operations (Section 3.3.2). A Slice configuration may include any number of coordinators, each managing a subset of the files (Section 4.2).

2.3 File Managers

File management functions above the network storage array are split across two classes of file managers. Each class governs functions that are common to any file server; the architecture separates them to distribute the request load and allow implementations specialized for each request class.

- *Directory servers* handle name space operations, e.g., to *create*, *remove*, or *lookup* files and directories by symbolic name; they manage directories and mappings from names to identifiers and attributes for each file or directory.
- *Small-file servers* handle *read* and *write* operations on small files and the initial segments of large files (Section 3.1).

Slice file managers are *dataless*; all of their state is backed by the network storage array. Their role is to aggregate their structures into larger storage objects backed by the storage nodes, and to provide memory and CPU resources to cache and manipulate those structures. In this way, the file managers can benefit from the parallel disk arms and high bandwidth of the storage array as more storage nodes are added.

The principle of dataless file managers also plays a key role in recovery. In addition to its backing objects, each manager journals its updates in a write-ahead log [10]; the system can recover the state of any manager from its backing objects together with its log. This allows fast failover, in which a surviving site assumes the role of a failed server, recovering its state from shared storage [12, 4, 24].

2.4 Summary

Interposed request routing in the Slice architecture yields three fundamental benefits:

- *Scalable file management with content-based request switching.* Slice distributes file service requests across a server ensemble. A good request switching scheme induces a balanced distribution of file objects and requests across servers, and improves locality in the request stream.
- *Direct storage access for high-volume I/O.* The μ proxy routes bulk I/O traffic directly to the network storage array, removing the file managers from the critical path. Separating requests in this fashion eliminates a key scaling barrier for conventional file services [8, 9]. At the same time, the small-file servers absorb and

aggregate I/O operations on small files, so there is no need for the storage nodes to handle small objects efficiently.

- *Compatibility with standard file system clients.* The μ proxy factors request routing policies out of the client-side file system code. This allows the architecture to leverage a minimal computing capability within the network elements to virtualize the storage protocol.

2.5 Related Work

A large number of systems have interposed new system functionality by “wrapping” an existing interface, including kernel system calls [14], internal interfaces [13], communication bindings [11], or messaging endpoints. The concept of a *proxy* mediating between clients and servers [23] is now common in distributed systems. We propose to mediate some storage functions by interposing on standard storage access protocols within the network elements. Network file services can benefit from this technique because they have well-defined protocols and a large installed base of clients and applications, many of which face significant scaling challenges today.

The Slice μ proxy routes file service requests based on their content. This is analogous to the HTTP content switching features offered by some network switch vendors (e.g., Alteon, Arrowpoint, F5), based in part on research demonstrating improved locality and load balancing for large Internet server sites [20]. Slice extends the content switching concept to a file system context.

A number of recent commercial and research efforts investigate techniques for building scalable storage systems for high-speed switched LAN networks. These systems are built from disks distributed through the network, and attached to dedicated servers [16, 24, 12], cooperating peers [4, 26], or the network itself [8, 9]. We separate these systems into two broad groups.

The first group separates file managers (e.g., the name service) from the block storage service, as in Slice. This separation was first proposed for the Cambridge Universal File Server [6]. Subsequent systems adopted this separation to allow bulk I/O to bypass file managers [7, 12], and it is now a basic tenet of research in network-attached storage devices including the CMU NASD work on devices for secure storage objects [8, 9]. Slice shows how to incorporate placement and routing functions essential for this separation into a new filesystem structure for network-attached storage. The CMU NASD

project integrated similar functions into network file system clients [9]; the Slice model decouples these functions, preserving compatibility with existing clients. In addition, Slice extends the NASD project approach to support scalable file management as well as high-bandwidth I/O for large files.

A second group of scalable storage systems layers the file system functions above a network storage volume using a *shared disk* model. Policies for striping, redundancy, and storage site selection are specified on a volume basis; cluster nodes coordinate their accesses to the shared storage blocks using an ownership protocol. This approach has been used with both log-structured (Zebra [12] and xFS [4]) and conventional (Frangipani/Petal [16, 24] and GFS [21]) file system organizations. The cluster may be viewed as “serverless” if all nodes are trusted and have direct access to the shared disk, or alternatively the entire cluster may act as a file server to untrusted clients using a standard network file protocol, with all I/O passing through the cluster nodes as they mediate access to the disks.

The key benefits of Slice request routing apply equally to these shared disk systems when untrusted clients are present. First, request routing is a key to incorporating secure network-attached block storage, which allows untrusted clients to address storage objects directly without compromising the integrity of the file system. That is, a μ proxy could route bulk I/O requests directly to the devices, yielding a more scalable system that preserves compatibility with standard clients and allows per-file policies for block placement, parity or replication, prefetching, etc. Second, request routing enhances locality in the request stream to the file servers, improving cache effectiveness and reducing block contention among the servers.

The shared disk model is used in many commercial systems, which increasingly interconnect storage devices and servers with dedicated Storage Area Networks (SANs), e.g., FibreChannel. This paper explores storage request routing for Internet networks, but the concepts are equally applicable in SANs.

Our proposal to separate small-file I/O from the request stream is similar in concept to the Amoeba Bullet Server [25], a specialized file server that optimizes small files. As described in Section 4.4, the prototype small-file server draws on techniques from the Bullet Server, FFS fragments [19], and SquidMLA [18], a Web proxy server that maintains a user-level “filesystem” of small cached Web pages.

3 Request Routing Policies

This section explains the structure of the μ proxy and the request routing schemes used in the Slice prototype. The purpose is to illustrate concretely the request routing policies enabled by the architecture, and the implications of those policies for the way the servers interact to maintain and recover consistent file system states. We use the NFS V3 protocol as a reference point because it is widely understood and our prototype supports it.

The μ proxy intercepts NFS requests addressed to virtual NFS servers, and routes the request to a physical server by applying a function to the request type and arguments. It then rewrites the IP address and port to redirect the request to the selected server. When a response arrives, the μ proxy rewrites the source address and port before forwarding it to the client, so the response appears to originate from the virtual NFS server.

The request routing functions must permit reconfiguration to add or remove servers, while minimizing state requirements in the μ proxy. The μ proxy directs most requests by extracting relevant fields from the request, perhaps hashing to combine multiple fields, and interpreting the result as a logical server site ID for the request. It then looks up the corresponding physical server in a compact routing table. Multiple logical sites may map to the same physical server, leaving flexibility for reconfiguration (Section 3.3.1). The routing tables constitute soft state; the mapping is determined externally, so the μ proxy never modifies the tables.

The μ proxy examines up to four fields of each request, depending on the policies configured:

- *Request type.* Routing policies are keyed by the NFS request type, so the μ proxy may employ different policies for different functions. Table 1 lists the important NFS request groupings discussed in this paper.
- *File handle.* Each NFS request targets a specific file or directory, named by a unique identifier called a *file handle* (or *fhandle*). Although NFS fhandles are opaque to the client, their structure can be known to the μ proxy, which acts as an extension of the service. Directory servers encode a *fileID* in each fhandle, which the μ proxies extract as a routing key.
- *Read/write offset.* NFS I/O operations specify the range of offsets covered by each *read* and

write. The μ proxy uses these fields to select the server or storage node for the data.

- *Name component.* NFS name space requests include a symbolic name component in their arguments (see Table 1). A key challenge for scaling file management is to obtain a balanced distribution of these requests. This is particularly important for *name-intensive* workloads with small files and heavy *create/lookup/remove* activity, as often occurs in Internet services for mail, news, message boards, and Web access.

We now outline some μ proxy policies that use these fields to route specific request groups.

3.1 Block I/O

Request routing for *read/write* requests have two goals: separate small-file *read/write* traffic from bulk I/O, and decluster the blocks of large files across the storage nodes for the desired access properties (e.g., high bandwidth or a specified level of redundancy). We address each in turn.

When small-file servers are configured, the prototype's routing policy defines a fixed *threshold offset* (e.g., 64KB); the μ proxy directs I/O requests below the threshold to a small-file server selected from the request fhandle. The threshold offset is necessary because the size of each file may change at any time. Thus the small-file servers also receive a subset of the I/O requests on large files; they receive *all* I/O below the threshold, even if the target file is large. In practice, large files have little impact on the small-file servers because there tends to be a small number of these files, even if they make up a large share of the stored bytes. Similarly, large file I/O below the threshold is limited by the bandwidth of the small-file server, but this affects only the first *threshold* bytes, and becomes progressively less significant as the file grows.

The μ proxy redirects I/O traffic above the threshold directly to the network storage array, using some placement policy to select the storage site(s) for each block. A simple option is to employ static striping and placement functions that compute on the block offset and/or fileID. More flexible placement policies would allow the μ proxy to consider other factors, e.g., load conditions on the network or storage nodes, or file attributes encoded in the fhandle. To generalize to more flexible placement policies, Slice optionally records block locations in per-file block maps managed by the block service coordinators. The μ proxies interact with the coordinators

Name Space Operations	
<code>lookup(dir, name)</code> returns (fhandle, attr)	Look up a name in <i>dir</i> , return handle and attributes.
<code>create(dir, name)</code> returns (fhandle, attr)	Create a file/directory and update the parent entry/link count and modify timestamp.
<code>mkdir(dir, name)</code> returns (fhandle, attr)	
<code>remove(dir, name), rmdir(dir, name)</code>	Remove a file/directory or hard link and update the parent entry/link count and modify timestamp.
<code>link(olddir, oldname, newdir, newname)</code> returns (fhandle, attr)	Create a new name for a file, update the file link count, and update modify timestamps on the file and <i>newdir</i> .
<code>rename(olddir, oldname, newdir, newname)</code> returns (fhandle, attr)	Rename an existing file or hard link; update the link count and modify timestamp on both the old and new parent.
Attribute Operations	
<code>getattr(object)</code> returns (attr)	Retrieve the attributes of a file or directory.
<code>setattr(object, attr)</code>	Modify the attributes of a file or directory, and update its modify timestamp.
I/O Operations	
<code>read(file, offset, len)</code> returns (data, attr)	Read data from a file, updating its access timestamp.
<code>write(file, offset, len)</code> returns (data, attr)	Write data to a file, updating its modify timestamp.
Directory Retrieval	
<code>readdir(dir, cookie)</code> returns (entries, cookie)	Read some or all of the entries in a directory.

Table 1: Some important Network File System (NFS) protocol operations.

to fetch and cache fragments of the block maps as they handle I/O operations on files.

As one example of an attribute-based policy, Slice supports a *mirrored striping* policy that replicates each block of a mirrored file on multiple storage nodes, to tolerate failures up to the replication degree. Mirroring consumes more storage and network bandwidth than striping with parity, but it is simple and reliable, avoids the overhead of computing and updating parity, and allows load-balanced reads [5, 16].

3.2 Name Space Operations

Effectively distributing name space requests presents different challenges from I/O request routing. Name operations involve more computation, and name entries may benefit more from caching because they tend to be relatively small and fragmented. Moreover, directories are frequently shared. Directory servers act as synchronization points to preserve integrity of the name space, e.g., to prevent clients from concurrently creating a file with the same name, or removing a directory while a name create is in progress.

A simple approach to scaling a file service is to partition the name space into a set of volumes, each managed by a single server. Unfortunately, this VOLUME PARTITIONING strategy compromises transparency and increases administrative overhead in two ways. First, volume boundaries are visible to clients as *mount points*, and naming operations such as *link* and *rename* cannot cross volume boundaries. Sec-

ond, the system develops imbalances if volume loads grow at different rates, requiring intervention to repartition the name space. This may be visible to users through name changes to existing directories.

An important goal of name management in Slice is to automatically distribute the load of a single file volume across multiple servers, without imposing user-visible volume boundaries. We propose two alternative name space routing policies to achieve this goal. MKDIR SWITCHING yields balanced distributions when the average number of active directories is large relative to the number of directory server sites, but it binds large directories to a single server. For workloads with very large directories, NAME HASHING yields probabilistically balanced request distributions independent of workload. The cost of this effectiveness is that more operations cross server boundaries, increasing the cost and complexity of coordination among the directory servers (Section 4.3).

MKDIR SWITCHING works as follows. In most cases, the μ proxy routes name space operations to the directory server that manages the parent directory; the μ proxy identifies this server by indexing its routing table with the fileID from the parent directory fhandle in the request (refer to Table 1). On a *mkdir* request, the μ proxy decides with probability p to redirect the request to a different directory server, placing the new directory — and its descendents — on a different site from the parent directory. The policy uniquely selects the new server by hashing on the parent fhandle and the symbolic name of the

new directory; this guarantees that races over name manipulation involve at most two sites. Reducing directory affinity by increasing p makes the policy more aggressive in distributing name entries across sites; this produces a more balanced load, but more operations involve multiple sites. Section 5 presents experimental data illustrating this tradeoff.

NAME HASHING extends this approach by routing *all* name space operations using a hash on the name component and its position in the directory tree, as given by the parent directory fhandle. This approach represents the entire volume name space as a unified global hash table distributed among the directory servers. It views directories as distributed collections of name entries, rather than as files accessed as a unit. Conflicting operations on any given name entry (e.g., *create/create*, *create/remove*, *remove/lookup*) always hash to the same server, where they serialize on the shared hash chain. Operations on different entries in the same directory (e.g., *create*, *remove*, *lookup*) may proceed in parallel at multiple sites. For good performance, NAME HASHING requires sufficient memory to keep the hash chains memory-resident, since the hashing function sacrifices locality in the hash chain accesses. Also, *read-dir* operations span multiple sites; this is the right behavior for large directories, but it increases *read-dir* costs for small directories.

3.3 Storage Service Structure

Request routing policies impact storage service structure. The primary challenges are coordination and recovery to maintain a consistent view of the file volume across all servers, and reconfiguration to add or remove servers within each class.

Most of the routing policies outlined above are independent of whether small files and name entries are bound to the server sites that create them. One option is for the servers to share backing objects from a *shared disk* using a block ownership protocol (see Section 2.5); in this case, the role of the μ proxy is to enhance locality in the request stream to each server. Alternatively, the system may use *fixed placement* in which items are controlled by their create sites unless reconfiguration or failover causes them to move; with this approach backing storage objects may be private to each site, even if they reside on shared network storage. Fixed placement stresses the role of the request routing policy in the placement of new name entries or data items. The next two subsections discuss reconfiguration and recovery issues for the Slice architecture with respect to these structural alternatives.

3.3.1 Reconfiguration

Consider the problem of reconfiguration to add or remove file managers, i.e., directory servers, small-file servers, or map coordinators. For requests routed by keying on the fileID, the system updates μ proxy routing tables to change the binding from fileIDs to physical servers if servers join or depart the ensemble. To keep the tables compact, Slice maps the fileID to a smaller logical server ID before indexing the table. The number of logical servers defines the size of the routing tables and the minimal granularity for rebalancing. The μ proxy's copy of the routing table is a "hint" that may become stale during reconfiguration; the μ proxy may load new tables lazily from an external source, assuming that servers can identify misdirected requests.

This approach generalizes to policies in which the logical server ID is derived from a hash that includes other request arguments, as in the NAME HASHING approach. For NAME HASHING systems and other systems with fixed placement, the reconfiguration procedure must move logical servers from one physical server to another. One approach is for each physical server to use multiple backing objects, one for each hosted logical server, and reconfigure by reassigning the binding of physical servers to backing objects in the shared network storage array. Otherwise, reconfiguration must copy data from one backing object to another. In general, an ensemble with N servers must move $1/N$ th of its data to rebalance after adding or losing a physical server [15].

3.3.2 Atomicity and Recovery

File systems have strong integrity requirements and frequent updates; the system must preserve their integrity through failures and concurrent operations. The focus on request routing naturally implies that the multiple servers must manage distributed state.

File managers prepare for recovery by generating a write-ahead log in shared storage. For systems that use the shared-disk model without fixed placement, all operations execute at a single manager site, and it is necessary and sufficient for the system to provide locking and recovery procedures for the shared disk blocks [24]. For systems with fixed placement, servers do not share blocks directly, but some operations must update state at multiple sites through a peer-peer protocol. Thus there is no need for distributed locking or recovery of individual blocks, but the system must coordinate logging and recovery across sites, e.g., using two-phase commit.

For MKDIR SWITCHING, the operations that update multiple sites are those involving the “orphaned” directories that were placed on different sites from their parents. These operations include the redirected *mkdirs* themselves, associated *rmdirs*, and any *rename* operations involving the orphaned entries. Since these operations are relatively infrequent, as determined by the redirection probability parameter p , it is acceptable to perform a full two-phase commit as needed to guarantee their atomicity on systems with fixed placement. However, NAME HASHING requires fixed placement — unless the directory servers support fine-grained distributed caching — and any name space update involves multiple sites with probability $(N - 1)/N$ or higher. While it is possible to reduce commit costs by logging asynchronously and coordinating rollback, this approach weakens failure properties because recently completed operations may be lost in a failure.

Shared network storage arrays present their own atomicity and recovery challenges. In Slice, the block service coordinators preserve atomicity of operations involving multiple storage nodes, including mirrored striping, *truncate/remove*, and NFS V3 write commitment (*commit*). Amiri et al. [1] addresses atomicity and concurrency control issues for shared storage arrays; the Slice coordinator protocol complements [1] with an intention logging protocol for atomic filesystem operations [2]. The basic protocol is as follows. At the start of the operation, the μ proxy sends to the coordinator an *intention* to perform the operation. The coordinator logs the intention to stable storage. When the operation completes, the μ proxy notifies the coordinator with a *completion* message, asynchronously clearing the intention. If the coordinator does not receive the completion within some time bound, it probes the participants to determine if the operation completed, and initiates recovery if necessary. A failed coordinator recovers by scanning its intentions log, completing or aborting operations in progress at the time of the failure. In practice, the protocol eliminates some message exchanges and log writes from the critical path of most common-case operations by piggybacking messages, leveraging the NFS V3 *commit* semantics, and amortizing intention logging costs across multiple operations.

4 Implementation

The Slice prototype is a set of loadable kernel modules for the FreeBSD operating system. The prototype includes a μ proxy implemented as a packet

filter below the Internet Protocol (IP) stack, and kernel modules for the basic server classes: block storage service and block storage coordinator, directory server, and small-file server. A given server node may be configured for any subset of the Slice server functions, and each function may be present at an arbitrary number of nodes. The following subsections discuss each element of the Slice prototype in more detail.

4.1 The μ proxy

The Slice μ proxy is a loadable packet filter module that intercepts packets exchanged with registered NFS virtual server endpoints. The module is configurable to run as an intermediary at any point in the network between a client and the server ensemble, preserving compatibility with NFS clients. Our premise is that the functions of the μ proxy are simple enough to integrate more tightly with the network switching elements, enabling wire-speed request routing. The μ proxy may also be configured below the IP stack on each client node, to avoid the store-and-forward delays imposed by host-based intermediaries in our prototype.

The μ proxy is a nonblocking state machine with soft state consisting of pending request records and routing tables for I/O redirection, MKDIR SWITCHING, and NAME HASHING, as described in Section 3. The prototype statically configures the policies and table sizes for name space operations and small-file I/O; it does not yet detect and refresh stale routing tables for reconfiguration. These policies use the MD5 [22] hash function; we determined empirically that MD5 yields a combination of balanced distribution and low cost that is superior to competing hash functions available to us. For *reads* and *writes* beyond the threshold offset the μ proxy may use either a static block placement policy or a local cache of per-file block maps supplied by a block service coordinator (see Section 4.2).

The μ proxy also maintains a cache over file attribute blocks returned in NFS responses from the servers. Directory servers maintain the authoritative attributes for files; the system must keep these attributes current to reflect I/O traffic to the block storage nodes, which affects the modify time, access time, and/or size attributes of the target file. The μ proxy updates these attributes in its cache as each operation completes, and returns a complete set of attributes to the client in each response (some clients depend on this behavior, although the NFS specification does not require it). The μ proxy generates an NFS *setattr* operation to push modified at-

tributes back to the directory server when it evicts attributes from its cache, or when it intercepts an NFS V3 write *commit* request from the client. Most clients issue *commit* requests for modified files from a periodic system update daemon, and when a user process calls *fsync* or *close* on a modified file.

The prototype may yield weaker attribute consistency than some NFS implementations. First, attribute timestamps are no longer assigned at a central site; we rely on the Network Time Protocol (NTP) to keep clocks synchronized across the system. Most NFS installations already use NTP to allow consistent assignment and interpretation of timestamps across multiple servers and clients. Second, a *read* or an uncommitted *write* is not guaranteed to update the attribute timestamps if the μ proxy fails and loses its state. In the worst case an uncommitted *write* might complete at a storage node but not affect the modify time at all (if the client also fails before reissuing the write). The NFS V3 specification permits this behavior: uncommitted writes may affect any subset of the modified data or attributes. Third, although the attribute timestamps cached and returned by each μ proxy are always current with respect to operations from clients bound to that μ proxy, they may drift beyond the “three second window” that is the de facto standard in NFS implementations for concurrently shared files. We consider this to be acceptable since NFS V3 offers no firm consistency guarantees for concurrently shared files anyway. Note, however, that NFS V4 proposes to support consistent file sharing through a leasing mechanism similar to NQ-NFS [17]; it will then be sufficient for the μ proxy to propagate file attributes when a client renews or relinquishes a lease for the file. The current μ proxy bounds the drift by writing back modified attributes at regular intervals.

Since the μ proxy modifies the contents of request and response packets, it must update the UDP or TCP checksums to match the new packet data. The prototype μ proxy recomputes checksums incrementally, generalizing a technique used in other packet rewriting systems. The μ proxy’s differential checksum code is derived from the FreeBSD implementation of Network Address Translation (NAT). The cost of incremental checksum adjustment is proportional to the number of modified bytes and is independent of the total size of the message. It is efficient because the μ proxy rewrites at most the source or destination address and port number, and in some cases certain fields of the file attributes.

4.2 Block Storage Service

The Slice block storage servers use a kernel module that exports disks to the network. The storage nodes serve a flat space of storage objects named by unique identifiers; storage is addressed by *(object, logicalblock)*, with physical allocation controlled by the storage node software as described in Section 2.2. The key operations are a subset of NFS, including *read*, *write*, *commit*, and *remove*. The storage nodes accept NFS file handles as object identifiers, using an external hash to map them to storage objects. Our current prototype uses the Fast File System (FFS) as a storage manager within each storage node. The storage nodes prefetch sequential files up to 256 KB beyond the current access, and also leverage FFS write clustering.

The block storage service includes a coordinator implemented as an extension to the storage node module. Each coordinator manages a set of files, selected by fileID. The coordinator maintains optional per-file block maps giving the storage site for each logical block of the file; these maps are used for dynamic I/O routing policies (Section 3.1). The coordinator also implements the intention logging protocol to preserve failure atomicity for file accesses involving multiple storage sites (Section 3.3.2), including remove/truncate, consistent write commitment, and mirrored writes, as described in [2]. The coordinator backs its intentions log and block maps within the block storage service using a static placement function. A more failure-resilient implementation would employ redundancy across storage nodes.

4.3 Directory Servers

Our directory server implementations use fixed placement and support both the NAME HASHING and MKDIR SWITCHING policies. The directory servers store directory information as webs of linked fixed-size cells representing name entries and file attributes, allocated from memory zones backed by the block storage service. These cells are indexed by hash chains keyed by an MD5 hash fingerprint on the parent file handle and name. The directory servers place keys in each newly minted file handle, allowing them to locate any resident cell if presented with an fhandle or an *(fhandle, name)* pair. Attribute cells may include a remote key to reference an entry on another server, enabling cross-site links in the directory structure. Thus the name entries and attribute cells for a directory may be distributed arbitrarily across the servers, making it possible to support both NAME HASHING and MKDIR SWITCHING policies easily within the same code base.

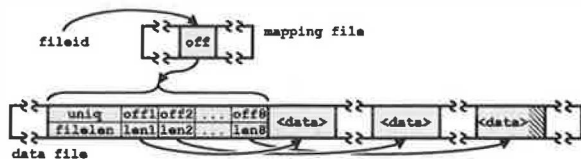


Figure 2: Small-file server data structures.

Given the distribution of entries across directory servers, some NFS operations involve multiple sites. The μ proxy interacts with a single site for each request. Directory servers use a simple peer-peer protocol to update link counts for *create/link/remove* and *mkdir/rmdir* operations that cross sites, and to follow cross-site links for *lookup*, *getattr/setattr*, and *readdir*. For NAME HASHING we implemented *rename* as a *link* followed by a *remove*.

Support for recovery and reconfiguration is incomplete in our prototype. Directory servers log their updates, but the recovery procedure itself is not implemented, nor is the support for shifting ownership of blocks and cells across servers.

4.4 Small-file Servers

The small-file server is implemented by a module that manages each file as a sequence of 8KB logical blocks. Figure 2 illustrates the key data structures and their use for a *read* or *write* request. The locations for each block are given by a per-file map record. The server accesses this record by indexing an on-disk map descriptor array using the fileID from the fhandle. Like the directory server, storage for small-file data is allocated from zones backed by objects in the block storage service.

Each map record gives a fixed number of (*offset, length*) pairs mapping 8KB file extents to regions within a backing object. Each logical block may have less than the full 8KB of physical space allocated for it; physical storage for a block rounds the space required up to the next power of two to simplify space management. New files or writes to empty segments are allocated space according to best fit, or if no good fragment is free, a new region is allocated at the end of the backing storage object. The best-fit variable fragment approach is similar to SquidMLA [18].

This structure allows efficient space allocation and supports file growth. For example, a 8300 byte file would consume only 8320 bytes of physical storage space, 8192 bytes for the first block, and 128 for the remaining 108 bytes. Under a create-heavy workload, the small-file allocation policy lays out data on

backing objects sequentially, batching newly created files into a single stream for efficient disk writes. The small-file servers comply with the NFS V3 *commit* specification for writes below the threshold offset.

Map records and data from the small-file server backing objects are cached in the kernel file buffer cache. This structure performs well if file accesses and the assignment of fileIDs show good locality. In particular, if the directory servers assign fileIDs with good spatial locality, and if files created together are accessed together, then the cost of reading the map records is amortized across multiple files whose records fit in a single block.

5 Performance

This section presents experimental results from the Slice prototype to show the overheads and scaling properties of the interposed request routing architecture. We use synthetic benchmarks to stress different aspects of the system, then evaluate whole-system performance using the industry-standard SPECsfs97 workload.

The storage nodes for the test ensemble are Dell PowerEdge 4400s with a 733 MHz Pentium-III Xeon CPU, 256MB RAM, and a ServerWorks LE chipset. Each storage node has eight 18GB Seagate Cheetah drives (ST318404LC) connected to a dual-channel Ultra-160 SCSI controller. Servers and clients are 450 MHz Pentium-III PCs with 512MB RAM and Asus P2B motherboards using a 440BX chipset. The machines are linked by a Gigabit Ethernet network with Alteon ACEnic 710025 adapters and a 32-port Extreme Summit-7i switch. The switch and adapters use 9KB (“Jumbo”) frames; the adapters run locally modified firmware that supports header splitting for NFS traffic. The adapters occupy a 64-bit/66 MHz PCI slot on the Dell 4400s, and a 32-bit/33 MHz PCI slot on the PCs. All kernels are built from the same FreeBSD 4.0 source pool.

	single client	saturation
read	62.5 MB/s	437 MB/s
write	38.9 MB/s	479 MB/s
read-mirrored	52.9 MB/s	222 MB/s
write-mirrored	32.2 MB/s	251 MB/s

Table 2: Bulk I/O bandwidth in the test ensemble.

Read/write performance. Table 2 shows raw read and write bandwidth for large files. Each test (*dd*) issues *read* or *write* system calls on a 1.25 GB file in a Slice volume mounted with a 32KB NFS

block size and a read-ahead depth of four blocks. The μ proxies use a static I/O routing function to stripe large-file data across the storage array. We measure sequential access bandwidth for unmirrored files and mirrored files with two replicas.

The left column of Table 2 shows the I/O bandwidth driven by a single PC client. Writes saturate the client CPU below 40 MB/s, the maximum bandwidth achievable through the FreeBSD NFS/UDP client stack in this configuration. We modified the FreeBSD client for zero-copy reading, allowing higher bandwidth with lower CPU utilization; in this case, performance is limited by a prefetch depth bound in FreeBSD. Mirroring degrades read bandwidth because the client μ proxies alternate between the two mirrors to balance the load, leaving some prefetched data unused on the storage nodes. Mirroring degrades write bandwidth because the client host writes to both mirrors.

The right column of Table 2 shows the aggregate bandwidth delivered to eight clients, saturating the storage node I/O systems. Each storage node sources reads to the network at 55 MB/s and sinks writes at 60 MB/s. While the Cheetah drives each yield 33 MB/s of raw bandwidth, achievable disk bandwidth is below 75 MB/s per node because the 4400 backplane has a single SCSI channel for all of its internal drive bays, and the FreeBSD 4.0 driver runs the channel in Ultra-2 mode because it does not yet support Ultra-160.

Operation	CPU
Packet interception	0.7%
Packet decode	4.1%
Redirection/rewriting	0.5%
Soft state logic	0.8%

Table 3: μ proxy CPU cost for 6250 packets/second.

Overhead of the μ proxy. The interposed request routing architecture is sensitive to the costs to intercept and redirect file service protocol packets. Table 3 summarizes the CPU overheads for a client-based μ proxy under a synthetic benchmark that stresses name space operations, which place the highest per-packet loads on the μ proxy. The benchmark repeatedly unpacks (*untar*) a set of zero-length files in a directory tree that mimics the FreeBSD source distribution. Each file create generates seven NFS operations: *lookup*, *access*, *create*, *getattr*, *lookup*, *setattr*, *setattr*. We used *iprobe* (Instruction Probe), an on-line profiling tool for Alpha-based systems, to measure the μ proxy CPU cost on

a 500 MHz Compaq 21264 client (4MB L2). This *untar* workload generates mixed NFS traffic at a rate of 3125 request/response pairs per second.

The client spends 6.1% of its CPU cycles in the μ proxy. Redirection replaces the packet destination and/or ports and restores the checksum as described in Section 4.1, consuming a modest 0.5% of CPU time. The cost of managing soft state for attribute updates and response pairing accounts for 0.8%. The most significant cost is the 4.1% of CPU time spent decoding the packets to prepare for rewriting. Nearly half of the cost is to locate the offsets of the NFS request type and arguments; NFS V3 and ONC RPC headers each include variable-length fields (e.g., access groups and the NFS V3 file handle) that increase the decoding overhead. Minor protocol changes could reduce this complexity. While this complexity affects the cost to implement the μ proxy in network elements, it does not limit the scalability of the Slice architecture.

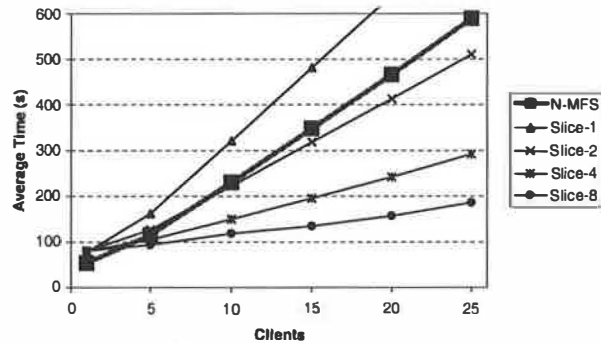


Figure 3: Directory service scaling.

Directory service scaling. We used the name-intensive *untar* benchmark to evaluate scalability of the prototype directory service using the NAME HASHING and MKDIR SWITCHING policies. For MKDIR SWITCHING we chose $p = 1/N$, i.e., the μ proxy redirects $1/N$ th of the *mkdir* requests to distribute the directories across the N server sites. In this test, a variable number of client processes execute the *untar* benchmark on five client PCs. Each process creates 36,000 files and directories generating a total of 250,000 NFS operations. For this experiment, in which the name space spans many directories, MKDIR SWITCHING and NAME HASHING perform identically.

Figure 3 shows the average total latency perceived by each client process as a function of the number of processes. We use multiple client nodes to avoid client saturation, and vary the number of directory servers; each line labeled “Slice- N ” has N

PCs acting as directory servers. For comparison, the N-MFS line measures an NFS server exporting a memory-based file system (FreeBSD MFS). MFS initially performs better due to Slice logging and update traffic, but the MFS server's CPU quickly saturates with more clients. In contrast, the Slice request routing schemes spread the load among multiple directory servers, and both schemes show good scaling behavior with more servers. Each server saturates at 6000 ops/s generating about 0.5 MB/s of log traffic.

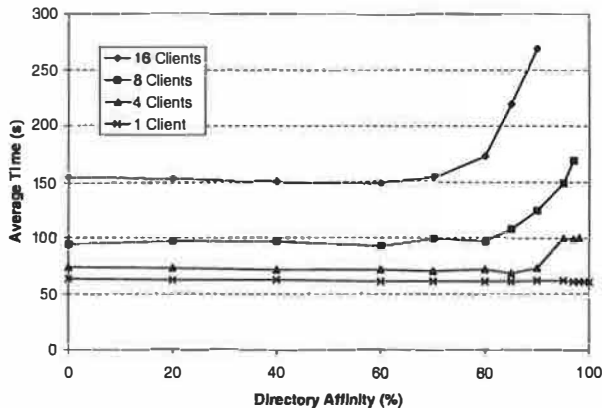


Figure 4: Impact of affinity for MKDIR SWITCHING.

Figure 4 shows the effect of varying directory affinity ($1 - p$) for MKDIR SWITCHING under the name-intensive *untar* workload. The X-axis gives the probability $1 - p$ that a new directory is placed on the same server as its parent; the Y-axis shows the average *untar* latency observed by the clients. This test uses four client nodes hosting one, four, eight, or sixteen client processes against four directory servers. For light workloads, latency is unaffected by affinity, since a single server can handle the load. For heavier workloads, increasing directory affinity along the X-axis initially yields a slight improvement as the number of cross-server operations declines. Increasing affinity toward 100% ultimately degrades performance due to load imbalances. This simple experiment indicates that MKDIR SWITCHING can produce even distributions while redirecting fewer than 20% of directory create requests. A more complete study is needed to determine the best parameters under a wider range of workloads.

Overall performance and scalability. We now report results from SPECsfs97, the industry-standard benchmark for network-attached storage. SPECsfs97 runs as a group of workload generator processes that produce a realistic mix of NFS V3 requests, check the responses against the NFS stan-

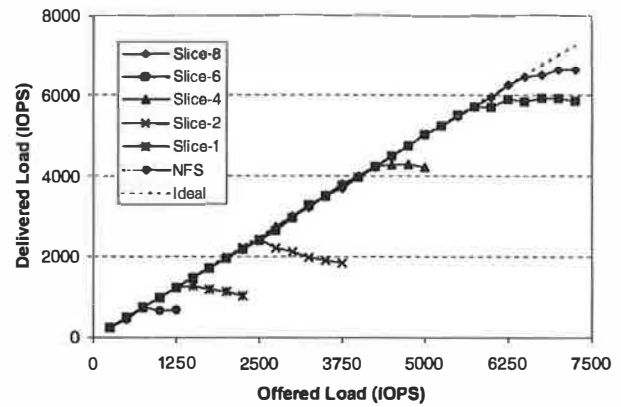


Figure 5: SPECsfs97 throughput at saturation.

dard, and measure latency and delivered throughput in I/O operations per second (IOPS). SPECsfs is designed to benchmark servers but not clients; it sends and receives NFS packets from user space without exercising the client kernel NFS stack. SPECsfs is a demanding, industrial-strength, self-scaling benchmark. We show results as evidence that the prototype is fully functional, complies with the NFS V3 standard, and is independent of any client NFS implementation, and to give a basis for judging prototype performance and scalability against commercial-grade servers.

The SPECsfs file set is skewed heavily toward small files: 94% of files are 64 KB or less. Although small files account for only 24% of the total bytes accessed, most SPECsfs I/O requests target small files; the large files serve to “pollute” the disks. Thus saturation throughput is determined largely by the number of disk arms. The Slice configurations for the SPECsfs experiments use a single directory server, two small-file servers, and a varying number of storage nodes. Figures 5 and 6 report results; lines labeled “Slice- N ” use N storage nodes.

Figure 5 gives delivered throughput for SPECsfs97 in IOPS as a function of offered load. As a baseline, the graph shows the 850 IOPS saturation point of a single FreeBSD 4.0 NFS server on a Dell 4400 exporting its disk array as a single volume (using the CCD disk concatenator). Slice-1 yields higher throughput than the NFS configuration due to faster directory operations, but throughput under load is constrained by the disk arms. The results show that Slice throughput scales with larger numbers of storage nodes, up to 6600 IOPS for eight storage nodes with a total of 64 disks.

Figure 6 gives average request latency as a function of delivered throughput. Latency jumps are evi-

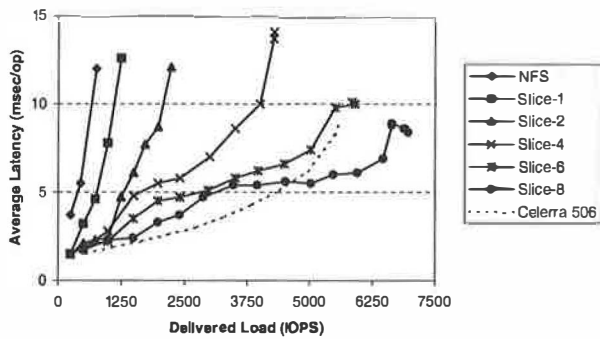


Figure 6: SPECsfs97 latency.

dent in the Slice results as the ensemble overflows its 1 GB cache on the small-file servers, but the prototype delivers acceptable latency at all workload levels up to saturation. For comparison, we include vendor-reported results from *spec.org* for a recent (4Q99) commercial server, the EMC Celerra File Server Cluster Model 506. The Celerra 506 uses 32 Cheetah drives for data and has 4 GB of cache. EMC Celerra is an industry-leading product: it delivers better latency and better throughput than the Slice prototype in the nearest equivalent configuration (Slice-4 with 32 drives), as well as better reliability through its use of RAID with parity. What is important is that the interposed request routing technique allows Slice to scale to higher IOPS levels by adding storage nodes and/or file manager nodes to the LAN. Celerra and other commercial storage servers are also expandable, but the highest IOPS ratings are earned by systems using a VOLUME PARTITIONING strategy to distribute load within the server. For example, this Celerra 506 exports eight separate file volumes. The techniques introduced in this paper allow high throughputs without imposing volume boundaries; all of the Slice configurations serve a single unified volume.

6 Conclusion

This paper explores interposed request routing in Slice, a new architecture for scalable network-attached storage. Slice interposes a simple redirecting μ proxy along the network path between the client and an ensemble of storage nodes and file managers. The μ proxy virtualizes a client/server file access protocol (e.g., NFS) by applying configurable request routing policies to distribute data and requests across the ensemble. The ensemble nodes cooperate to provide a unified, scalable file service.

The Slice μ proxy distributes requests by request type and by target object, combining functional de-

composition and data decomposition of the request traffic. We describe two policies for distributing name space requests, MKDIR SWITCHING and NAME HASHING, and demonstrate their potential to automatically distribute name space load across servers. These techniques complement simple grouping and striping policies to distribute file access load.

The Slice prototype delivers high bandwidth and high request throughput on an industry-standard NFS benchmark, demonstrating scalability of the architecture and prototype. Experiments with a simple μ proxy packet filter show the feasibility of incorporating the request routing features into network elements. The prototype demonstrates that the interposed request routing architecture enables incremental construction of powerful distributed storage services while preserving compatibility with standard file system clients.

Availability. For more information please visit the Web site at <http://www.cs.duke.edu/ari/slice>.

Acknowledgements. This work benefited from discussions with many people, including Khalil Amiri, Mike Burrows, Carla Ellis, Garth Gibson, Dan Muntz, Tom Rodeheffer, Chandu Thekkath, John Wilkes, Ken Yocum, and Zheng Zhang. Andrew Gallatin assisted with hardware and software in numerous ways. We thank the anonymous reviewers and our shepherd, Timothy Roscoe, for useful critiques and suggestions.

References

- [1] Khalil Amiri, Garth Gibson, and Richard Golding. Highly concurrent shared storage. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, April 2000.
- [2] Darrell C. Anderson and Jeffrey S. Chase. Failure-atomic file access in an interposed network storage system. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC)*, August 2000.
- [3] David Anderson. Object Based Storage Devices: A command set proposal. Technical report, National Storage Industry Consortium, October 1999.
- [4] Tom Anderson, Michael Dahlin, J. Neefe, David Patterson, Drew Roselli, and Randy Wang. Serverless network file systems. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 109–126, December 1995.
- [5] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David A. Patterson, and Katherine Yelick. Cluster I/O with River: Making the fast case com-

- mon. In *I/O in Parallel and Distributed Systems (IOPADS)*, May 1999.
- [6] Andrew D. Birrell and Roger M. Needham. A universal file server. *IEEE Transactions on Software Engineering*, SE-6(5):450–453, September 1980.
- [7] Luis-Felipe Cabrera and Darrell D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computing Systems*, 4(4):405–436, Fall 1991.
- [8] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka. File server scaling with network-attached secure disks. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, volume 25,1 of *Performance Evaluation Review*, pages 272–284, New York, June 15–18 1997. ACM Press.
- [9] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.
- [10] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, pages 155–162, 1987.
- [11] Graham Hamilton, Michael L. Powell, and James J. Mitchell. Subcontract: A flexible base for distributed programming. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 69–79, December 1993.
- [12] John H. Hartman and John K. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems*, 13(3):274–310, August 1995.
- [13] John S. Heidemann and Gerald J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.
- [14] Michael B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the Fourteenth Symposium on Operating Systems Principles*, pages 80–93, December 1993.
- [15] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the Twenty-Ninth ACM Symposium on Theory of Computing*, pages 654–663, El Paso, May 1997.
- [16] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, Cambridge, MA, October 1996.
- [17] R. Macklem. Not quite NFS, soft cache consistency for NFS. In *USENIX Association Conference Proceedings*, pages 261–278, January 1994.
- [18] Carlos Maltzahn, Kathy Richardson, and Dirk Grunwald. Reducing the disk I/O of web proxy server caches. In *USENIX Annual Technical Conference*, June 1999.
- [19] Marshall K. McKusick, William Joy, Samuel Leffler, and Robert Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.
- [20] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenopoe, and Erich Nahum. Locality-aware request distribution in cluster-based network servers. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.
- [21] Kenneth Preslan, Andrew Barry, Jonathan Brasow, Grant Erickson, Erling Nygaard, Christopher Sabol, Steven Soltis, David Teigland, and Matthew O’Keefe. A 64-bit, shared disk file system for Linux. In *Sixteenth IEEE Mass Storage Systems Symposium*, March 1999.
- [22] Ron L. Rivest. RFC 1321: The MD5 message-digest algorithm, April 1992. <ftp://ftp.internic.net/rfc/rfc1321.txt>.
- [23] Marc Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *Proceedings of the Sixth International Conference on Distributed Computing Systems*, May 1986.
- [24] Chandu Thekkath, Tim Mann, and Ed Lee. Frangipani: A scalable distributed file system. In *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224–237, October 1997.
- [25] Robbert van Renesse, Andrew Tanenbaum, and Annita Wilschut. The design of a high-performance file server. In *The 9th International Conference on Distributed Computing Systems*, pages 22–27, Newport Beach, CA, June 1989. IEEE.
- [26] Geoff M. Voelker, Eric J. Anderson, Tracy Kimbrel, Michael J. Feeley, Jeffrey S. Chase, Anna R. Karlin, and Henry M. Levy. Implementing cooperative prefetching and caching in a globally-managed memory system. In *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS ’98)*, June 1998.

Proactive Recovery in a Byzantine-Fault-Tolerant System

Miguel Castro and Barbara Liskov

Laboratory for Computer Science,

Massachusetts Institute of Technology,

545 Technology Square, Cambridge, MA 02139

{castro,liskov}@lcs.mit.edu

Abstract

This paper describes an asynchronous state-machine replication system that tolerates Byzantine faults, which can be caused by malicious attacks or software errors. Our system is the first to recover Byzantine-faulty replicas proactively and it performs well because it uses symmetric rather than public-key cryptography for authentication. The recovery mechanism allows us to tolerate any number of faults over the lifetime of the system provided fewer than 1/3 of the replicas become faulty within a window of vulnerability that is small under normal conditions. The window may increase under a denial-of-service attack but we can detect and respond to such attacks. The paper presents results of experiments showing that overall performance is good and that even a small window of vulnerability has little impact on service latency.

1 Introduction

This paper describes a new system for asynchronous state-machine replication [17, 28] that offers both integrity and high availability in the presence of Byzantine faults. Our system is interesting for two reasons: it improves security by recovering replicas proactively, and it is based on symmetric cryptography, which allows it to perform well so that it can be used in practice to implement real services.

Our system continues to function correctly even when some replicas are compromised by an attacker; this is worthwhile because the growing reliance on online information services makes malicious attacks more likely and their consequences more serious. The system also survives nondeterministic software bugs and software bugs due to aging (e.g., memory leaks). Our approach improves on the usual technique of rebooting the system because it refreshes state automatically, staggers recovery so that individual replicas are highly unlikely to fail simultaneously, and has little impact on overall system performance. Section 4.7 discusses the types of faults tolerated by the system in more detail.

Because of recovery, our system can tolerate any number of faults over the lifetime of the system, provided fewer than 1/3 of the replicas become faulty within

a window of vulnerability. The best that could be guaranteed previously was correct behavior if fewer than 1/3 of the replicas failed during the lifetime of a system. Our previous work [6] guaranteed this and other systems [26, 16] provided weaker guarantees. Limiting the number of failures that can occur in a finite window is a synchrony assumption but such an assumption is unavoidable: since Byzantine-faulty replicas can discard the service state, we must bound the number of failures that can occur before recovery completes. But we require no synchrony assumptions to match the guarantee provided by previous systems. We compare our approach with other work in Section 7.

The window of vulnerability can be small (e.g., a few minutes) under normal conditions. Additionally, our algorithm provides *detection* of denial-of-service attacks aimed at increasing the window: replicas can time how long a recovery takes and alert their administrator if it exceeds some pre-established bound. Therefore, integrity can be preserved even when there is a denial-of-service attack.

The paper describes a number of new techniques needed to solve the problems that arise when providing recovery from Byzantine faults:

Proactive recovery. A Byzantine-faulty replica may appear to behave properly even when broken; therefore recovery must be proactive to prevent an attacker from compromising the service by corrupting 1/3 of the replicas without being detected. Our algorithm recovers replicas periodically independent of any failure detection mechanism. However a recovering replica may not be faulty and recovery must not cause it to become faulty, since otherwise the number of faulty replicas could exceed the bound required to provide safety. In fact, we need to allow the replica to continue participating in the request processing protocol while it is recovering, since this is sometimes required for it to complete the recovery.

Fresh messages. An attacker must be prevented from impersonating a replica that was faulty after it recovers. This can happen if the attacker learns the keys used to authenticate messages. Furthermore even if messages are signed using a secure cryptographic co-processor, an attacker might be able to authenticate bad messages while it controls a faulty replica; these messages could be replayed later to compromise safety. To solve this problem, we define a notion of authentication *freshness*

This research was supported by DARPA under contract F30602-98-1-0237 monitored by the Air Force Research Laboratory.

and replicas reject messages that are not fresh. However, this leads to a further problem, since replicas may be unable to prove to a third party that some message they received is authentic (because it may no longer be fresh). All previous state-machine replication algorithms [26, 16], including the one we described in [6], relied on such proofs. Our current algorithm does not, and this has the added advantage of enabling the use of symmetric cryptography for authentication of all protocol messages. This eliminates most use of public-key cryptography, the major performance bottleneck in previous systems.

Efficient state transfer. State transfer is harder in the presence of Byzantine faults and efficiency is crucial to enable frequent recovery with little impact on performance. To bring a recovering replica up to date, the state transfer mechanism checks the local copy of the state to determine which portions are both up-to-date and not corrupt. Then, it must ensure that any missing state it obtains from other replicas is correct. We have developed an efficient hierarchical state transfer mechanism based on hash chaining and incremental cryptography [1]; the mechanism tolerates Byzantine-faults and state modifications while transfers are in progress.

Our algorithm has been implemented as a generic program library with a simple interface. This library can be used to provide Byzantine-fault-tolerant versions of different services. The paper describes experiments that compare the performance of a replicated NFS implemented using the library with an unreplicated NFS. The results show that the performance of the replicated system without recovery is close to the performance of the unreplicated system. They also show that it is possible to recover replicas frequently to achieve a small window of vulnerability in the normal case (2 to 10 minutes) with little impact on service latency.

The rest of the paper is organized as follows. Section 2 presents our system model and lists our assumptions; Section 3 states the properties provided by our algorithm; and Section 4 describes the algorithm. Our implementation is described in Section 5 and some performance experiments are presented in Section 6. Section 7 discusses related work. Our conclusions are presented in Section 8.

2 System Model and Assumptions

We assume an asynchronous distributed system where nodes are connected by a network. The network may fail to deliver messages, delay them, duplicate them, or deliver them out of order.

We use a Byzantine failure model, i.e., faulty nodes may behave arbitrarily, subject only to the restrictions mentioned below. We allow for a very strong adversary that can coordinate faulty nodes, delay communication, inject messages into the network, or delay correct nodes in order to cause the most damage to the replicated service. We do assume that the adversary cannot delay correct nodes indefinitely.

We use cryptographic techniques to establish session keys, authenticate messages, and produce digests. We use

the SFS [21] implementation of a Rabin-Williams public-key cryptosystem with a 1024-bit modulus to establish 128-bit session keys. All messages are then authenticated using message authentication codes (MACs) [2] computed using these keys. Message digests are computed using MD5 [27].

We assume that the adversary (and the faulty nodes it controls) is computationally bound so that (with very high probability) it is unable to subvert these cryptographic techniques. For example, the adversary cannot forge signatures or MACs without knowing the corresponding keys, or find two messages with the same digest. The cryptographic techniques we use are thought to have these properties.

Previous Byzantine-fault tolerant state-machine replication systems [6, 26, 16] also rely on the assumptions described above. We require no additional assumptions to match the guarantees provided by these systems, i.e., to provide safety if less than 1/3 of the replicas become faulty during the lifetime of the system. To tolerate more faults we need additional assumptions: we must mutually authenticate a faulty replica that recovers to the other replicas, and we need a reliable mechanism to trigger periodic recoveries. These could be achieved by involving system administrators in the recovery process, but such an approach is impractical given our goal of recovering replicas frequently. Instead, we rely on the following assumptions:

Secure Cryptography. Each replica has a secure cryptographic co-processor, e.g., a Dallas Semiconductors iButton, or the security chip in the motherboard of the IBM PC 300PL. The co-processor stores the replica's private key, and can sign and decrypt messages without exposing this key. It also contains a true random number generator, e.g., based on thermal noise, and a counter that never goes backwards. This enables it to append random numbers or the counter to messages it signs.

Read-Only Memory. Each replica stores the public keys for other replicas in some memory that survives failures without being corrupted (provided the attacker does not have physical access to the machine). This memory could be a portion of the flash BIOS. Most motherboards can be configured such that it is necessary to have physical access to the machine to modify the BIOS.

Watchdog Timer. Each replica has a *watchdog timer* that periodically interrupts processing and hands control to a *recovery monitor*, which is stored in the read-only memory. For this mechanism to be effective, an attacker should be unable to change the rate of watchdog interrupts without physical access to the machine. Some motherboards and extension cards offer the watchdog timer functionality but allow the timer to be reset without physical access to the machine. However, this is easy to fix by preventing write access to control registers unless some jumper switch is closed.

These assumptions are likely to hold when the attacker does not have physical access to the replicas, which we expect to be the common case. When they fail we can fall back on system administrators to perform recovery.

Note that all previous proactive security algorithms [24, 13, 14, 3, 10] assume the entire program run by a replica is in read-only memory so that it cannot be modified by an attacker. Most also assume that there are authenticated channels between the replicas that continue to work even after a replica recovers from a compromise. These assumptions would be sufficient to implement our algorithm but they are less likely to hold in practice. We only require a small monitor in read-only memory and use the secure co-processors to establish new session keys between the replicas after a recovery.

The only work on proactive security that does not assume authenticated channels is [3], but the best that a replica can do when its private key is compromised in their system is alert an administrator. Our *secure cryptography* assumption enables automatic recovery from most failures, and secure co-processors with the properties we require are now readily available, e.g., IBM is selling PCs with a cryptographic co-processor in the motherboard at essentially no added cost.

We also assume clients have a secure co-processor; this simplifies the key exchange protocol between clients and replicas but it could be avoided by adding an extra round to this protocol.

3 Algorithm Properties

Our algorithm is a form of *state machine* replication [17, 28]: the service is modeled as a state machine that is replicated across different nodes in a distributed system. The algorithm can be used to implement any replicated service with a *state* and some *operations*. The operations are not restricted to simple reads and writes; they can perform arbitrary computations.

The service is implemented by a set of replicas \mathcal{R} and each replica is identified using an integer in $\{0, \dots, |\mathcal{R}| - 1\}$. Each replica maintains a copy of the service state and implements the service operations. For simplicity, we assume $|\mathcal{R}| = 3f + 1$ where f is the maximum number of replicas that may be faulty. Service clients and replicas are non-faulty if they follow the algorithm and if no attacker can impersonate them (e.g., by forging their MACs).

Like all state machine replication techniques, we impose two requirements on replicas: they must start in the same state, and they must be *deterministic* (i.e., the execution of an operation in a given state and with a given set of arguments must always produce the same result). We can handle some common forms of non-determinism using the technique we described in [6].

Our algorithm ensures safety for an execution provided at most f replicas become faulty within a window of vulnerability of size T_v . Safety means that the replicated service satisfies linearizability [12, 5]: it behaves like a centralized implementation that executes operations atomically one at a time. Our algorithm provides safety regardless of how many faulty clients are using the service (even if they collude with faulty replicas).

We will discuss the window of vulnerability further in Section 4.7.

The algorithm also guarantees liveness: non-faulty clients eventually receive replies to their requests provided (1) at most f replicas become faulty within the window of vulnerability T_v ; and (2) denial-of-service attacks do not last forever, i.e., there is some unknown point in the execution after which all messages are delivered (possibly after being retransmitted) within some constant time d , or all non-faulty clients have received replies to their requests. Here, d is a constant that depends on the timeout values used by the algorithm to refresh keys, and trigger view-changes and recoveries.

4 Algorithm

The algorithm works as follows. Clients send requests to execute operations to the replicas and all non-faulty replicas execute the same operations in the same order. Since replicas are deterministic and start in the same state, all non-faulty replicas send replies with identical results for each operation. The client waits for $f + 1$ replies from different replicas with the same result. Since at least one of these replicas is not faulty, this is the correct result of the operation.

The hard problem is guaranteeing that *all non-faulty replicas agree on a total order for the execution of requests despite failures*. We use a primary-backup mechanism to achieve this. In such a mechanism, replicas move through a succession of configurations called *views*. In a view one replica is the *primary* and the others are *backups*. We choose the primary of a view to be replica p such that $p = v \bmod |\mathcal{R}|$, where v is the view number and views are numbered consecutively.

The primary picks the ordering for execution of operations requested by clients. It does this by assigning a sequence number to each request. But the primary may be faulty. Therefore, the backups trigger *view changes* when it appears that the primary has failed to select a new primary. Viewstamped Replication [23] and Paxos [18] use a similar approach to tolerate benign faults.

To tolerate Byzantine faults, every step taken by a node in our system is based on obtaining a *certificate*. A certificate is a set of messages certifying some *statement* is correct and coming from different replicas. An example of a statement is: "the result of the operation requested by a client is r ".

The size of the set of messages in a certificate is either $f + 1$ or $2f + 1$, depending on the type of statement and step being taken. The correctness of our system depends on a certificate never containing more than f messages sent by faulty replicas. A certificate of size $f + 1$ is sufficient to prove that the statement is correct because it contains at least one message from a non-faulty replica. A certificate of size $2f + 1$ ensures that it will also be possible to convince other replicas of the validity of the statement even when f replicas are faulty.

Our earlier algorithm [6] used the same basic ideas but it did not provide recovery. Recovery complicates the

construction of certificates; if a replica collects messages for a certificate over a sufficiently long period of time it can end up with more than f messages from faulty replicas. We avoid this problem by introducing a notion of *freshness*; replicas reject messages that are not fresh. But this raises another problem: the view change protocol in [6] relied on the exchange of certificates between replicas and this may be impossible because some of the messages in a certificate may no longer be fresh. Section 4.5 describes a new view change protocol that solves this problem and also eliminates the need for expensive public-key cryptography.

To provide liveness with the new protocol, a replica must be able to fetch missing state that may be held by a single correct replica whose identity is not known. In this case, voting cannot be used to ensure correctness of the data being fetched and it is important to prevent a faulty replica from causing the transfer of unnecessary or corrupt data. Section 4.6 describes a mechanism to obtain missing messages and state that addresses these issues and that is efficient to enable frequent recoveries.

The sections below describe our algorithm. Sections 4.2 and 4.3, which explain normal-case request processing, are similar to what appeared in [6]. They are presented here for completeness and to highlight some subtle changes.

4.1 Message Authentication

We use MACs to authenticate all messages. There is a pair of session keys for each pair of replicas i and j : $k_{i,j}$ is used to compute MACs for messages sent from i to j , and $k_{j,i}$ is used for messages sent from j to i .

Some messages in the protocol contain a single MAC computed using UMAC32 [2]; we denote such a message as $\langle m \rangle_{\mu_{i,j}}$, where i is the sender j is the receiver and the MAC is computed using $k_{i,j}$. Other messages contain *authenticators*; we denote such a message as $\langle m \rangle_{\alpha_i}$, where i is the sender. An authenticator is a vector of MACs, one per replica j ($j \neq i$), where the MAC in entry j is computed using $k_{i,j}$. The receiver of a message verifies its authenticity by checking the corresponding MAC in the authenticator.

Replicas and clients refresh the session keys used to send messages to them by sending *new-key* messages periodically (e.g., every minute). The same mechanism is used to establish the initial session keys. The message has the form $\langle \text{NEW-KEY}, i, \dots, \{k_{j,i}\}_{\epsilon_j}, \dots, t \rangle_{\sigma_i}$. The message is signed by the secure co-processor (using the replica's private key) and t is the value of its counter; the counter is incremented by the co-processor and appended to the message every time it generates a signature. (This prevents suppress-replay attacks [11].) Each $k_{j,i}$ is the key replica j should use to authenticate messages it sends to i in the future; $k_{j,i}$ is encrypted by j 's public key, so that only j can read it. Replicas use timestamp t to detect spurious new-key messages: t must be larger than the timestamp of the last new-key message received from i .

Each replica shares a single secret key with each client; this key is used for communication in both

directions. The key is refreshed by the client periodically, using the new-key message. If a client neglects to do this within some system-defined period, a replica discards its current key for that client, which forces the client to refresh the key.

When a replica or client sends a new-key message, it discards all messages in its log that are not part of a complete certificate and it rejects any messages it receives in the future that are authenticated with old keys. This ensures that correct nodes only accept certificates with *equally fresh* messages, i.e., messages authenticated with keys created in the same refreshment phase.

4.2 Processing Requests

We use a three-phase protocol to atomically multicast requests to the replicas. The three phases are *pre-prepare*, *prepare*, and *commit*. The pre-prepare and prepare phases are used to totally order requests sent in the same view even when the primary, which proposes the ordering of requests, is faulty. The prepare and commit phases are used to ensure that requests that commit are totally ordered across views. Figure 1 shows the operation of the algorithm in the normal case of no primary faults.

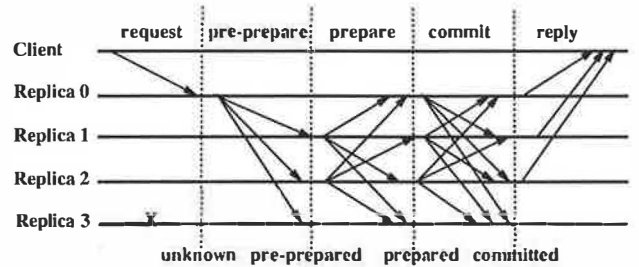


Figure 1: Normal Case Operation. Replica 0 is the primary, and replica 3 is faulty

Each replica stores the service state, a *log* containing information about requests, and an integer denoting the replica's current view. The log records information about the request associated with each sequence number, including its status; the possibilities are: *unknown* (the initial status), *pre-prepared*, *prepared*, and *committed*. Figure 1 also shows the evolution of the request status as the protocol progresses. We describe how to truncate the log in Section 4.3.

A client c requests the execution of state machine operation o by sending a $\langle \text{REQUEST}, o, t, c \rangle_{\alpha_c}$ message to the primary. Timestamp t is used to ensure *exactly-once* semantics for the execution of client requests [6].

When the primary p receives a request m from a client, it assigns a sequence number n to m . Then it multicasts a pre-prepare message with the assignment to the backups, and marks m as pre-prepared with sequence number n . The message has the form $\langle \langle \text{PRE-PREPARE}, v, n, d \rangle_{\alpha_p}, m \rangle$, where v indicates the view in which the message is being sent, and d is m 's digest.

Like pre-prepares, the prepare and commit messages

sent in the other phases also contain n and v . A replica only accepts one of these messages if it is in view v ; it can verify the authenticity of the message; and n is between a low water mark, h , and a high water mark, H . The last condition is necessary to enable garbage collection and prevent a faulty primary from exhausting the space of sequence numbers by selecting a very large one. We discuss how H and h advance in Section 4.3.

A backup i accepts the pre-prepare message provided (in addition to the conditions above): it has not accepted a pre-prepare for view v and sequence number n containing a different digest; it can verify the authenticity of m ; and d is m 's digest. If i accepts the pre-prepare, it marks m as pre-prepared with sequence number n , and enters the *prepare* phase by multicasting a $\langle \text{PREPARE}, v, n, d, i \rangle_{\alpha_i}$ message to all other replicas.

When replica i has accepted a certificate with a pre-prepare message and $2f$ prepare messages for the same sequence number n and digest d (each from a different replica including itself), it marks the message as *prepared*. The protocol guarantees that other non-faulty replicas will either prepare the same request or will not prepare any request with sequence number n in view v .

Replica i multicasts $\langle \text{COMMIT}, v, n, d, i \rangle_{\alpha_i}$ saying it prepared the request. This starts the commit phase. When a replica has accepted a certificate with $2f + 1$ commit messages for the same sequence number n and digest d from different replicas (including itself), it marks the request as *committed*. The protocol guarantees that the request is prepared with sequence number n in view v at $f + 1$ or more non-faulty replicas. This ensures information about committed requests is propagated to new views.

Replica i executes the operation requested by the client when m is committed with sequence number n and the replica has executed all requests with lower sequence numbers. This ensures that all non-faulty replicas execute requests in the same order as required to provide safety.

After executing the requested operation, replicas send a reply to the client c . The reply has the form $\langle \text{REPLY}, v, t, c, i, r \rangle_{\mu_{ic}}$ where t is the timestamp of the corresponding request, i is the replica number, and r is the result of executing the requested operation. This message includes the current view number v so that clients can track the current primary.

The client waits for a certificate with $f + 1$ replies from different replicas and with the same t and r , before accepting the result r . This certificate ensures that the result is valid. If the client does not receive replies soon enough, it broadcasts the request to all replicas. If the request is not executed, the primary will eventually be suspected to be faulty by enough replicas to cause a view change and select a new primary.

4.3 Garbage Collection

Replicas can discard entries from their log once the corresponding requests have been executed by at least $f + 1$ non-faulty replicas; this many replicas are needed

to ensure that the execution of that request will be known after a view change.

We can determine this condition by extra communication, but to reduce cost we do the communication only when a request with a sequence number divisible by some constant K (e.g., $K = 128$) is executed. We will refer to the states produced by the execution of these requests as *checkpoints*.

When replica i produces a checkpoint, it multicasts a $\langle \text{CHECKPOINT}, n, d, i \rangle_{\alpha_i}$ message to the other replicas, where n is the sequence number of the last request whose execution is reflected in the state and d is the digest of the state. A replica maintains several logical copies of the service state: the current state and some previous checkpoints. Section 4.6 describes how we manage checkpoints efficiently.

Each replica waits until it has a certificate containing $2f + 1$ valid checkpoint messages for sequence number n with the same digest d sent by different replicas (including possibly its own message). At this point, the checkpoint is said to be *stable* and the replica discards all entries in its log with sequence numbers less than or equal to n ; it also discards all earlier checkpoints.

The checkpoint protocol is used to advance the low and high water marks (which limit what messages will be added to the log). The low-water mark h is equal to the sequence number of the last stable checkpoint and the high water mark is $H = h + L$, where L is the log size. The log size is obtained by multiplying K by a small constant factor (e.g., 2) that is big enough so that replicas do not stall waiting for a checkpoint to become stable.

4.4 Recovery

The recovery protocol makes faulty replicas behave correctly again to allow the system to tolerate more than f faults over its lifetime. To achieve this, the protocol ensures that after a replica recovers it is running correct code; it cannot be impersonated by an attacker; and it has correct, up-to-date state.

Reboot. Recovery is proactive — it starts periodically when the watchdog timer goes off. The recovery monitor saves the replica's state (the log and the service state) to disk. Then it reboots the system with correct code and restarts the replica from the saved state. The correctness of the operating system and service code is ensured by storing them in a read-only medium (e.g., the Seagate Cheetah 18LP disk can be write protected by physically closing a jumper switch). Rebooting restores the operating system data structures and removes any Trojan horses.

After this point, the replica's code is correct and it did not lose its state. The replica must retain its state and use it to process requests even while it is recovering. This is vital to ensure both safety and liveness in the common case when the recovering replica is not faulty; otherwise, recovery could cause the $f + 1$ st fault. But if the recovering replica was faulty, the state may be corrupt and the attacker may forge messages because it

knows the MAC keys used to authenticate both incoming and outgoing messages. The rest of the recovery protocol solves these problems.

The recovering replica i starts by discarding the keys it shares with clients and it multicasts a new-key message to change the keys it uses to authenticate messages sent by the other replicas. This is important if i was faulty because otherwise the attacker could prevent a successful recovery by impersonating any client or replica.

Run estimation protocol. Next, i runs a simple protocol to estimate an upper bound, H_M , on the high-water mark that it would have in its log if it were not faulty. It discards any entries with greater sequence numbers to bound the sequence number of corrupt entries in the log.

Estimation works as follows: i multicasts a $\langle \text{QUERY-STABLE}, i, r \rangle_{\alpha_i}$ message to all the other replicas, where r is a random nonce. When replica j receives this message, it replies $\langle \text{REPLY-STABLE}, c, p, i, r \rangle_{\mu_{ji}}$, where c and p are the sequence numbers of the last checkpoint and the last request prepared at j respectively. i keeps retransmitting the query message and processing replies; it keeps the minimum value of c and the maximum value of p it received from each replica. It also keeps its own values of c and p .

The recovering replica uses the responses to select H_M as follows: $H_M = L + c_M$ where L is the log size and c_M is a value c received from replica j such that $2f$ replicas other than j reported values for c less than or equal to c_M and f replicas other than j reported values of p greater than or equal to c_M .

For safety, c_M must be greater than any stable checkpoint so that i will not discard log entries when it is not faulty. This is insured because if a checkpoint is stable it will have been created by at least $f + 1$ non-faulty replicas and it will have a sequence number less than or equal to any value of c that they propose. The test against p ensures that c_M is close to a checkpoint at some non-faulty replica since at least one non-faulty replica reports a p not less than c_M ; this is important because it prevents a faulty replica from prolonging i 's recovery. Estimation is live because there are $2f + 1$ non-faulty replicas and they only propose a value of c if the corresponding request committed and that implies that it prepared at at least $f + 1$ correct replicas.

After this point i participates in the protocol as if it were not recovering but it will not send any messages above H_M until it has a correct stable checkpoint with sequence number greater than or equal to H_M .

Send recovery request. Next i sends a recovery request with the form: $\langle \text{REQUEST}, \langle \text{RECOVERY}, H_M \rangle, t, i \rangle_{\sigma_i}$. This message is produced by the cryptographic co-processor and t is the co-processor's counter to prevent replays. The other replicas reject the request if it is a replay or if they accepted a recovery request from i recently (where recently can be defined as half of the watchdog period). This is important to prevent a denial-of-service attack where non-faulty replicas are kept busy executing recovery requests.

The recovery request is treated like any other request: it is assigned a sequence number n_R and it goes through the usual three phases. But when another replica executes the recovery request, it sends its own new-key message. Replicas also send a new-key message when they fetch missing state (see Section 4.6) and determine that it reflects the execution of a new recovery request. This is important because these keys are known to the attacker if the recovering replica was faulty. By changing these keys, we bound the sequence number of messages forged by the attacker that may be accepted by the other replicas — they are guaranteed not to accept forged messages with sequence numbers greater than the maximum high water mark in the log when the recovery request executes, i.e., $H_R = \lfloor n_R/K \rfloor \times K + L$.

The reply to the recovery request includes the sequence number n_R . Replica i uses the same protocol as the client to collect the correct reply to its recovery request but waits for $2f + 1$ replies. Then it computes its *recovery point*, $H = \max(H_M, H_R)$. It also computes a valid view (see Section 4.5); it retains its current view if there are $f + 1$ replies for views greater than or equal to it, else it changes to the median of the views in the replies.

Check and fetch state. While i is recovering, it uses the state transfer mechanism discussed in Section 4.6 to determine what pages of the state are corrupt and to fetch pages that are out-of-date or corrupt.

Replica i is *recovered* when the checkpoint with sequence number H is stable. This ensures that any state other replicas relied on i to have is actually held by $f + 1$ non-faulty replicas. Therefore if some other replica fails now, we can be sure the state of the system will not be lost. This is true because the estimation procedure run at the beginning of recovery ensures that while recovering i never sends bad messages for sequence numbers above the recovery point. Furthermore, the recovery request ensures that other replicas will not accept forged messages with sequence numbers greater than H .

Our protocol has the nice property that any replica knows that i has completed its recovery when checkpoint H is stable. This allows replicas to estimate the duration of i 's recovery, which is useful to detect denial-of-service attacks that slow down recovery with low false positives.

4.5 View Change Protocol

The view change protocol provides liveness by allowing the system to make progress when the current primary fails. The protocol must preserve safety: it must ensure that non-faulty replicas agree on the sequence numbers of committed requests across views. In addition, the protocol must provide liveness: it must ensure that non-faulty replicas stay in the same view long enough for the system to make progress, even in the face of a denial-of-service attack.

The new view change protocol uses the techniques described in [6] to address liveness but uses a different approach to preserve safety. Our earlier approach relied

on certificates that were valid indefinitely. In the new protocol, however, the fact that messages can become stale means that a replica cannot prove the validity of a certificate to others. Instead the new protocol relies on the group of replicas to validate each statement that some replica claims has a certificate. The rest of this section describes the new protocol.

Data structures. Replicas record information about what happened in earlier views. This information is maintained in two sets, the *PSet* and the *QSet*. A replica also stores the requests corresponding to the entries in these sets. These sets only contain information for sequence numbers between the current low and high water marks in the log; therefore only limited storage is required. The sets allow the view change protocol to work properly even when more than one view change occurs before the system is able to continue normal operation; the sets are usually empty while the system is running normally.

The *PSet* at replica i stores information about requests that have prepared at i in previous views. Its entries are tuples $\langle n, d, v \rangle$ meaning that a request with digest d prepared at i with number n in view v and no request prepared at i in a later view.

The *QSet* stores information about requests that have pre-prepared at i in previous views (i.e., requests for which i has sent a pre-prepare or prepare message). Its entries are tuples $\langle n, \{ \dots, \langle d_k, v_k \rangle, \dots \} \rangle$ meaning that for each k , v_k is the latest view in which a request pre-prepared with sequence number n and digest d_k at i .

View-change messages. View changes are triggered when the current primary is suspected to be faulty (e.g., when a request from a client is not executed after some period of time; see [6] for details). When a backup i suspects the primary for view v is faulty, it enters view $v+1$ and multicasts a $\langle \text{VIEW-CHANGE}, v+1, ls, C, P, Q, i \rangle_{\alpha_i}$ message to all replicas. Here ls is the sequence number of the latest stable checkpoint known to i ; C is a set of pairs with the sequence number and digest of each checkpoint stored at i ; and P and Q are sets containing a tuple for every request that is prepared or pre-prepared, respectively, at i . These sets are computed using the information in the log, the *PSet*, and the *QSet*, as explained in Figure 2. Once the view-change message has been sent, i stores P in *PSet*, Q in *QSet*, and clears its log. The computation bounds the size of each tuple in *QSet*; it retains only pairs corresponding to $f+2$ distinct requests (corresponding to possibly f messages from faulty replicas, one message from a good replica, and one special null message as explained below). Therefore the amount of storage used is bounded.

View-change-ack messages. Replicas collect view-change messages for $v+1$ and send acknowledgments for them to $v+1$'s primary, p . The acknowledgments have the form $\langle \text{VIEW-CHANGE-ACK}, v+1, i, j, d \rangle_{\mu_{ij}}$, where i is the identifier of the sender, d is the digest of the view-change message being acknowledged, and j is the replica that sent that view-change message. These acknowledgments allow the primary to prove authenticity of view-change messages sent by faulty replicas as explained later.

let v be the view before the view change, L be the size of the log, and h be the log's low water mark

```

for all  $n$  such that  $h < n \leq h+L$  do
  if request number  $n$  with digest  $d$  is prepared or
    committed in view  $v$  then
    add  $\langle n, d, v \rangle$  to  $P$ 
  else if  $\exists \langle n, d', v' \rangle \in PSet$  then
    add  $\langle n, d', v' \rangle$  to  $P$ 
  if request number  $n$  with digest  $d$  is pre-prepared,
    prepared or committed in view  $v$  then
    if  $\neg \exists \langle n, D \rangle \in QSet$  then
      add  $\langle n, \{ \langle d, v \rangle \} \rangle$  to  $Q$ 
    else if  $\exists \langle d, v' \rangle \in D$  then
      add  $\langle n, D \cup \{ \langle d, v \rangle \} - \{ \langle d, v' \rangle \} \rangle$  to  $Q$ 
    else if  $|D| > f+1$  then
      remove entry with lowest view number from  $D$ 
      add  $\langle n, D \cup \{ \langle d, v \rangle \} \rangle$  to  $Q$ 
    else if  $\exists \langle n, D \rangle \in QSet$  then
      add  $\langle n, D \rangle$  to  $Q$ 

```

Figure 2: Computing P and Q

New-view message construction. The new primary p collects view-change and view-change-ack messages (including messages from itself). It stores view-change messages in a set S . It adds a view-change message received from replica i to S after receiving $2f-1$ view-change-acks for i 's view-change message from other replicas. Each entry in S is for a different replica.

```

let  $D = \{ \langle n, d \rangle \mid \exists 2f+1 \text{ messages } m \in S : m.ls \leq n \wedge \exists f+1 \text{ messages } m \in S : \langle n, d \rangle \in m.C \}$ 
if  $\exists \langle h, d \rangle \in D : \forall \langle n', d' \rangle \in D : n' \leq h$  then
  select checkpoint with digest  $d$  and number  $h$ 
else exit
for all  $n$  such that  $h < n \leq h+L$  do
  A. if  $\exists m \in S$  with  $\langle n, d, v \rangle \in m.P$  that verifies:
    A1.  $\exists 2f+1$  messages  $m' \in S$ :
       $m'.ls < n \wedge m'.P$  has no entry for  $n$  or
       $\exists \langle n, d', v' \rangle \in m'.P : v' < v \vee (v' = v \wedge d' = d)$ 
    A2.  $\exists f+1$  messages  $m' \in S$ :
       $\exists \langle n, \{ \dots, \langle d', v' \rangle, \dots \} \rangle \in m'.Q : v' \geq v \wedge d' = d$ 
    A3. the primary has the request with digest  $d$ 
      then select the request with digest  $d$  for number  $n$ 
  B. else if  $\exists 2f+1$  messages  $m \in S$  such that
       $m.ls < n \wedge m.P$  has no entry for  $n$ 
      then select the null request for number  $n$ 

```

Figure 3: Decision procedure at the primary.

The new primary uses the information in S and the decision procedure sketched in Figure 3 to choose a checkpoint and a set of requests. This procedure runs each time the primary receives new information, e.g., when it adds a new message to S .

The primary starts by selecting the checkpoint that is going to be the starting state for request processing in the new view. It picks the checkpoint with the highest number h from the set of checkpoints that are known to be correct and that have numbers higher than the low

water mark in the log of at least $f + 1$ non-faulty replicas. The last condition is necessary for safety; it ensures that the ordering information for requests that committed with numbers higher than h is still available.

Next, the primary selects a request to pre-prepare in the new view for each sequence number between h and $h + L$ (where L is the size of the log). For each number n that was assigned to some request m that committed in a previous view, the decision procedure selects m to pre-prepare in the new view with the same number. This ensures safety because no distinct request can commit with that number in the new view. For other numbers, the primary may pre-prepare a request that was in progress but had not yet committed, or it might select a special *null* request that goes through the protocol as a regular request but whose execution is a no-op.

We now argue informally that this procedure will select the correct value for each sequence number. If a request m committed at some non-faulty replica with number n , it prepared at least $f + 1$ non-faulty replicas and the view-change messages sent by those replicas will indicate that m prepared with number n . Any set of at least $2f + 1$ view-change messages for the new view must include a message from one of the non-faulty replicas that prepared m . Therefore, the primary for the new view will be unable to select a different request for number n because no other request will be able to satisfy conditions A1 or B (in Figure 3).

The primary will also be able to make the right decision eventually: condition A1 will be satisfied because there are $2f + 1$ non-faulty replicas and non-faulty replicas never prepare different requests for the same view and sequence number; A2 is also satisfied since a request that prepares at a non-faulty replica pre-prepares at at least $f + 1$ non-faulty replicas. Condition A3 may not be satisfied initially, but the primary will eventually receive the request in a response to its status messages (discussed in Section 4.6). When a missing request arrives, this will trigger the decision procedure to run.

The decision procedure ends when the primary has selected a request for each number. This takes $O(L \times |\mathcal{R}|^3)$ local steps in the worst case but the normal case is much faster because most replicas propose identical values. After deciding, the primary multicasts a new-view message to the other replicas with its decision. The new-view message has the form $\langle \text{NEW-VIEW}, v + 1, \mathcal{V}, \mathcal{X} \rangle_{\alpha_p}$. Here, \mathcal{V} contains a pair for each entry in \mathcal{S} consisting of the identifier of the sending replica and the digest of its view-change message, and \mathcal{X} identifies the checkpoint and request values selected.

New-view message processing. The primary updates its state to reflect the information in the new-view message. It records all requests in \mathcal{X} as pre-prepared in view $v + 1$ in its log. If it does not have the checkpoint with sequence number h it also initiates the protocol to fetch the missing state (see Section 4.6.2). In any case the primary does not accept any prepare or commit messages with sequence number less than or equal to h and does not send any pre-prepare message with such a sequence number.

The backups for view $v + 1$ collect messages until they have a correct new-view message and a correct matching view-change message for each pair in \mathcal{V} . If some replica changes its keys in the middle of a view change, it has to discard all the view-change protocol messages it already received with the old keys. The message retransmission mechanism causes the other replicas to re-send these messages using the new keys.

If a backup did not receive one of the view-change messages for some replica with a pair in \mathcal{V} , the primary alone may be unable to prove that the message it received is authentic because it is not signed. The use of view-change-ack messages solves this problem. The primary only includes a pair for a view-change message in \mathcal{S} after it collects $2f - 1$ matching view-change-ack messages from other replicas. This ensures that at least $f + 1$ non-faulty replicas can vouch for the authenticity of every view-change message whose digest is in \mathcal{V} . Therefore, if the original sender of a view-change is uncooperative, the primary retransmits that sender's view-change message and the non-faulty backups retransmit their view-change-acks. A backup can accept a view-change message whose authenticator is incorrect if it receives f view-change-acks that match the digest and identifier in \mathcal{V} .

After obtaining the new-view message and the matching view-change messages, the backups check whether these messages support the decisions reported by the primary by carrying out the decision procedure in Figure 3. If they do not, the replicas move immediately to view $v + 2$. Otherwise, they modify their state to account for the new information in a way similar to the primary. The only difference is that they multicast a prepare message for $v + 1$ for each request they mark as pre-prepared. Thereafter, the protocol proceeds as described in Section 4.2.

The replicas use the status mechanism in Section 4.6 to request retransmission of missing requests as well as missing view-change, view-change acknowledgment, and new-view messages.

4.6 Obtaining Missing Information

This section describes the mechanisms for message retransmission and state transfer. The state transfer mechanism is necessary to bring replicas up to date when some of the messages they are missing were garbage collected.

4.6.1 Message Retransmission

We use a receiver-based recovery mechanism similar to SRM [8]: a replica i multicasts small *status* messages that summarize its state; when other replicas receive a status message they retransmit messages they have sent in the past that i is missing. Status messages are sent periodically and when the replica detects that it is missing information (i.e., they also function as negative acks).

If a replica j is unable to validate a status message, it sends its last new-key message to i . Otherwise, j sends messages it sent in the past that i may be missing. For

example, if i is in a view less than j 's, j sends i its latest view-change message. In all cases, j authenticates messages it retransmits with the latest keys it received in a new-key message from i . This is important to ensure liveness with frequent key changes.

Clients retransmit requests to replicas until they receive enough replies. They measure response times to compute the retransmission timeout and use a randomized exponential backoff if they fail to receive a reply within the computed timeout.

4.6.2 State Transfer

A replica may learn about a stable checkpoint beyond the high water mark in its log by receiving checkpoint messages or as the result of a view change. In this case, it uses the state transfer mechanism to fetch modifications to the service state that it is missing.

It is important for the state transfer mechanism to be efficient because it is used to bring a replica up to date during recovery, and we perform proactive recoveries frequently. The key issues to achieving efficiency are reducing the amount of information transferred and reducing the burden imposed on replicas. This mechanism must also ensure that the transferred state is correct. We start by describing our data structures and then explain how they are used by the state transfer mechanism.

Data Structures. We use hierarchical state partitions to reduce the amount of information transferred. The root partition corresponds to the entire service state and each non-leaf partition is divided into s equal-sized, contiguous sub-partitions. We call leaf partitions *pages* and interior partitions meta-data. For example, the experiments described in Section 6 were run with a hierarchy with four levels, s equal to 256, and 4KB pages.

Each replica maintains one logical copy of the partition tree for each checkpoint. The copy is created when the checkpoint is taken and it is discarded when a later checkpoint becomes stable. The tree for a checkpoint stores a tuple $\langle lm, d \rangle$ for each meta-data partition and a tuple $\langle lm, d, p \rangle$ for each page. Here, lm is the sequence number of the checkpoint at the end of the last checkpoint interval where the partition was modified, d is the digest of the partition, and p is the value of the page.

The digests are computed efficiently as follows. For a page, d is obtained by applying the MD5 hash function [27] to the string obtained by concatenating the index of the page within the state, its value of lm and p . For meta-data, d is obtained by applying MD5 to the string obtained by concatenating the index of the partition within its level, its value of lm , and the sum modulo a large integer of the digests of its sub-partitions. Thus, we apply AdHash [1] at each meta-data level. This construction has the advantage that the digests for a checkpoint can be obtained efficiently by updating the digests from the previous checkpoint incrementally.

The copies of the partition tree are logical because we use copy-on-write so that only copies of the tuples modified since the checkpoint was taken are stored. This

reduces the space and time overheads for maintaining these checkpoints significantly.

Fetching State. The strategy to fetch state is to recurse down the hierarchy to determine which partitions are out of date. This reduces the amount of information about (both non-leaf and leaf) partitions that needs to be fetched.

A replica i multicasts $\langle \text{FETCH}, l, x, lc, c, k, i \rangle_{\alpha_i}$ to all replicas to obtain information for the partition with index x in level l of the tree. Here, lc is the sequence number of the last checkpoint i knows for the partition, and c is either -1 or it specifies that i is seeking the value of the partition at sequence number c from replica k .

When a replica i determines that it needs to initiate a state transfer, it multicasts a fetch message for the root partition with lc equal to its last checkpoint. The value of c is non-zero when i knows the correct digest of the partition information at checkpoint c , e.g., after a view change completes i knows the digest of the checkpoint that propagated to the new view but might not have it. i also creates a new (logical) copy of the tree to store the state it fetches and initializes a table \mathcal{LC} in which it stores the number of the latest checkpoint reflected in the state of each partition in the new tree. Initially each entry in the table will contain lc .

If $\langle \text{FETCH}, l, x, lc, c, k, i \rangle_{\alpha_i}$ is received by the designated replier, k , and it has a checkpoint for sequence number c , it sends back $\langle \text{META-DATA}, c, l, x, P, k \rangle$, where P is a set with a tuple $\langle x', lm, d \rangle$ for each sub-partition of (l, x) with index x' , digest d , and $lm > lc$. Since i knows the correct digest for the partition value at checkpoint c , it can verify the correctness of the reply without the need for voting or even authentication. This reduces the burden imposed on other replicas.

The other replicas only reply to the fetch message if they have a stable checkpoint greater than lc and c . Their replies are similar to k 's except that c is replaced by the sequence number of their stable checkpoint and the message contains a MAC. These replies are necessary to guarantee progress when replicas have discarded a specific checkpoint requested by i .

Replica i retransmits the fetch message (choosing a different k each time) until it receives a valid reply from some k or $f + 1$ equally fresh responses with the same sub-partition values for the same sequence number cp (greater than lc and c). Then, it compares its digests for each sub-partition of (l, x) with those in the fetched information; it multicasts a fetch message for sub-partitions where there is a difference, and sets the value in \mathcal{LC} to c (or cp) for the sub-partitions that are up to date. Since i learns the correct digest of each sub-partition at checkpoint c (or cp) it can use the optimized protocol to fetch them.

The protocol recurses down the tree until i sends fetch messages for out-of-date pages. Pages are fetched like other partitions except that meta-data replies contain the digest and last modification sequence number for the page rather than sub-partitions, and the designated replier sends back $\langle \text{DATA}, x, p \rangle$. Here, x is the page index and p is the page value. The protocol imposes little overhead on other replicas; only one replica replies with the full

page and it does not even need to compute a MAC for the message since i can verify the reply using the digest it already knows.

When i obtains the new value for a page, it updates the state of the page, its digest, the value of the last modification sequence number, and the value corresponding to the page in \mathcal{LC} . Then, the protocol goes up to its parent and fetches another missing sibling. After fetching all the siblings, it checks if the parent partition is *consistent*. A partition is consistent up to sequence number c if c is the minimum of all the sequence numbers in \mathcal{LC} for its sub-partitions, and c is greater than or equal to the maximum of the last modification sequence numbers in its sub-partitions. If the parent partition is not consistent, the protocol sends another fetch for the partition. Otherwise, the protocol goes up again to its parent and fetches missing siblings.

The protocol ends when it visits the root partition and determines that it is consistent for some sequence number c . Then the replica can start processing requests with sequence numbers greater than c .

Since state transfer happens concurrently with request execution at other replicas and other replicas are free to garbage collect checkpoints, it may take some time for a replica to complete the protocol, e.g., each time it fetches a missing partition, it receives information about yet a later modification. This is unlikely to be a problem in practice (this intuition is confirmed by our experimental results). Furthermore, if the replica fetching the state ever is actually needed because others have failed, the system will wait for it to catch up.

4.7 Discussion

Our system ensures safety and liveness for an execution τ provided at most f replicas become faulty within a window of vulnerability of size $T_v = 2T_k + T_r$. The values of T_k and T_r are characteristic of each execution τ and unknown to the algorithm. T_k is the maximum key refreshment period in τ for a non-faulty node, and T_r is the maximum time between when a replica fails and when it recovers from that fault in τ .

The message authentication mechanism from Section 4.1 ensures non-faulty nodes only accept certificates with messages generated within an interval of size at most $2T_k$.¹ The bound on the number of faults within T_v ensures there are never more than f faulty replicas within any interval of size at most $2T_k$. Therefore, safety and liveness are provided because non-faulty nodes never accept certificates with more than f bad messages.

We have little control over the value of T_v because T_r may be increased by a denial-of-service attack, but we have good control over T_k and the maximum time between watchdog timeouts, T_w , because their values are determined by timer rates, which are quite stable. Setting these timeout values involves a tradeoff between

¹It would be T_k except that during view changes replicas may accept messages that are claimed authentic by $f + 1$ replicas without directly checking their authentication token.

security and performance: small values improve security by reducing the window of vulnerability but degrade performance by causing more frequent recoveries and key changes. Section 6 analyzes this tradeoff.

The value of T_w should be set based on R_n , the time it takes to recover a non-faulty replica under normal load conditions. There is no point in recovering a replica when its previous recovery has not yet finished; and we stagger the recoveries so that no more than f replicas are recovering at once, since otherwise service could be interrupted even without an attack. Therefore, we set $T_w = 4 \times s \times R_n$. Here, the factor 4 accounts for the staggered recovery of $3f + 1$ replicas f at a time, and s is a safety factor to account for benign overload conditions (i.e., no attack).

Another issue is the bound f on the number of faults. Our replication technique is not useful if there is a strong positive correlation between the failure probabilities of the replicas; the probability of exceeding the bound may not be lower than the probability of a single fault in this case. Therefore, it is important to take steps to increase diversity. One possibility is to have diversity in the execution environment: the replicas can be administered by different people; they can be in different geographic locations; and they can have different configurations (e.g., run different combinations of services, or run schedulers with different parameters). This improves resilience to several types of faults, for example, attacks involving physical access to the replicas, administrator attacks or mistakes, attacks that exploit weaknesses in other services, and software bugs due to race conditions. Another possibility is to have software diversity; replicas can run different operating systems and different implementations of the service code. There are several independent implementations available for operating systems and important services (e.g. file systems, data bases, and WWW servers). This improves resilience to software bugs and attacks that exploit software bugs.

Even without taking any steps to increase diversity, our proactive recovery technique increases resilience to nondeterministic software bugs, to software bugs due to aging (e.g., memory leaks), and to attacks that take more time than T_v to succeed. It is possible to improve security further by exploiting software diversity across recoveries. One possibility is to restrict the service interface at a replica after its state is found to be corrupt. Another potential approach is to use obfuscation and randomization techniques [7, 9] to produce a new version of the software each time a replica is recovered. These techniques are not very resilient to attacks but they can be very effective when combined with proactive recovery because the attacker has a bounded time to break them.

5 Implementation

We implemented the algorithm as a library with a very simple interface (see Figure 4). Some components of the library run on clients and others at the replicas.

On the client side, the library provides a procedure

```

Client:
int Byz_init_client(char *conf);
int Byz_invoke(Byz_req *req, Byz_rep *rep, bool read_only);

Server:
int Byz_init_replica(char *conf, char *mem, int size, UC exec);
void Byz_modify(char *mod, int size);

Server upcall:
int execute(Byz_req *req, Byz_rep *rep, int client);

```

Figure 4: The replication library API.

to initialize the client using a configuration file, which contains the public keys and IP addresses of the replicas. The library also provides a procedure, *invoke*, that is called to cause an operation to be executed. This procedure carries out the client side of the protocol and returns the result when enough replicas have responded.

On the server side, we provide an initialization procedure that takes as arguments a configuration file with the public keys and IP addresses of replicas and clients, the region of memory where the application state is stored, and a procedure to execute requests. When our system needs to execute an operation, it makes an upcall to the *execute* procedure. This procedure carries out the operation as specified for the application, using the application state. As the application performs the operation, each time it is about to modify the application state, it calls the *modify* procedure to inform us of the locations about to be modified. This call allows us to maintain checkpoints and compute digests efficiently as described in Section 4.6.2.

6 Performance Evaluation

This section has two parts. First, it presents results of experiments to evaluate the benefit of eliminating public-key cryptography from the critical path. Then, it presents an analysis of the cost of proactive recoveries.

6.1 Experimental Setup

All experiments ran with four replicas. Four replicas can tolerate one Byzantine fault; we expect this reliability level to suffice for most applications. Clients and replicas ran on Dell Precision 410 workstations with Linux 2.2.16-3 (uniprocessor). These workstations have a 600 MHz Pentium III processor, 512 MB of memory, and a Quantum Atlas 10K 18WLS disk. All machines were connected by a 100 Mb/s switched Ethernet and had 3Com 3C905B interface cards. The switch was an Extreme Networks Summit48 V4.1. The experiments ran on an isolated network.

The interval between checkpoints, K , was 128 requests, which causes garbage collection to occur several times in each experiment. The size of the log, L , was 256. The state partition tree had 4 levels, each internal node had 256 children, and the leaves had 4 KB.

6.2 The cost of Public-Key Cryptography

To evaluate the benefit of using MACs instead of public key signatures, we implemented BFT-PK. Our previous algorithm [6] relies on the extra power of digital signatures to authenticate pre-prepare, prepare, checkpoint, and view-change messages but it can be easily modified to use MACs to authenticate other messages. To provide a fair comparison, BFT-PK is identical to the BFT library but it uses public-key signatures to authenticate these four types of messages. We ran a micro benchmark, and a file system benchmark to compare the performance of services implemented with the two libraries. There were no view changes, recoveries or key changes in these experiments.

6.2.1 Micro-Benchmark

The micro-benchmark compares the performance of two implementations of a simple service: one implementation uses BFT-PK and the other uses BFT. This service has no state and its operations have arguments and results of different sizes but they do nothing. We also evaluated the performance of NO-REP: an implementation of the service using UDP with no replication. We ran experiments to evaluate the latency and throughput of the service. The comparison with NO-REP shows the worst case overhead for our library; in real services, the relative overhead will be lower due to computation or I/O at the clients and servers.

Table 1 reports the latency to invoke an operation when the service is accessed by a single client. The results were obtained by timing a large number of invocations in three separate runs. We report the average of the three runs. The standard deviations were always below 0.5% of the reported value.

system	0/0	0/4	4/0
BFT-PK	59368	59761	59805
BFT	431	999	1046
NO-REP	106	625	630

Table 1: Micro-benchmark: operation latency in microseconds. Each operation type is denoted by a/b , where a and b are the sizes of the argument and result in KB.

BFT-PK has two signatures in the critical path and each of them takes 29.4 ms to compute. The algorithm described in this paper eliminates the need for these signatures. As a result, BFT is between 57 and 138 times faster than BFT-PK. BFT's latency is between 60% and 307% higher than NO-REP because of additional communication and computation overhead. For read-only requests, BFT uses the optimization described in [6] that reduces the slowdown for operations 0/0 and 0/4 to 93% and 25%, respectively.

We also measured the overhead of replication at the client. BFT increases CPU time relative to NO-REP by up to a factor of 5, but the CPU time at the client is only between 66 and 142 μ s per operation. BFT also increases the number of bytes in Ethernet packets that are sent or

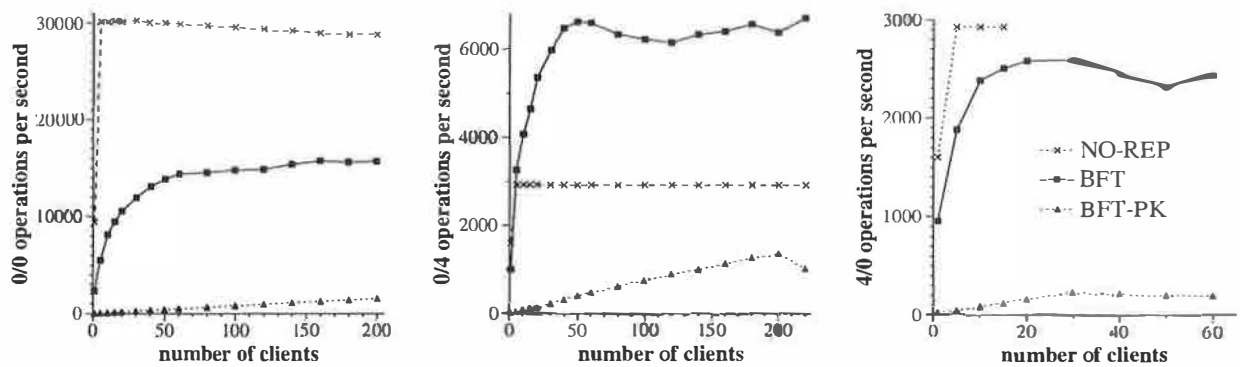


Figure 5: Micro-benchmark: throughput in operations per second.

received by the client: 405% for the 0/0 operation but only 12% for the other operations.

Figure 5 compares the throughput of the different implementations of the service as a function of the number of clients. The client processes were evenly distributed over 5 client machines² and each client process invoked operations synchronously, i.e., it waited for a reply before invoking a new operation. Each point in the graph is the average of at least three independent runs and the standard deviation for all points was below 4% of the reported value (except that it was as high as 17% for the last four points in the graph for BFT-PK operation 4/0). There are no points with more than 15 clients for NO-REP operation 4/0 because of lost request messages; NO-REP uses UDP directly and does not retransmit requests.

The throughput of both replicated implementations increases with the number of concurrent clients because the library implements batching [4]. Batching inlines several requests in each pre-prepare message to amortize the protocol overhead. BFT-PK performs 5 to 11 times worse than BFT because signing messages leads to a high protocol overhead and there is a limit on how many requests can be inlined in a pre-prepare message.

The bottleneck in operation 0/0 is the server's CPU; BFT's maximum throughput is 53% lower than NO-REP's due to extra messages and cryptographic operations that increase the CPU load. The bottleneck in operation 4/0 is the network; BFT's throughput is within 11% of NO-REP's because BFT does not consume significantly more network bandwidth in this operation. BFT achieves a maximum aggregate throughput of 26 MB/s in operation 0/4 whereas NO-REP is limited by the link bandwidth (approximately 12 MB/s). The throughput is better in BFT because of an optimization that we described in [6]: each client chooses one replica randomly; this replica's reply includes the 4 KB but the replies of the other replicas only contain small digests. As a result, clients obtain the large replies in parallel from different replicas. We refer the reader to [4] for a detailed analysis of these latency and throughput results.

²Two client machines had 700 MHz PIIIs but were otherwise identical to the other machines.

6.2.2 File System Benchmarks

We implemented the Byzantine-fault-tolerant NFS service that was described in [6]. The next set of experiments compares the performance of two implementations of this service: BFS, which uses BFT, and BFS-PK, which uses BFT-PK.

The experiments ran the modified Andrew benchmark [25, 15], which emulates a software development workload. It has five phases: (1) creates subdirectories recursively; (2) copies a source tree; (3) examines the status of all the files in the tree without examining their data; (4) examines every byte of data in all the files; and (5) compiles and links the files. Unfortunately, Andrew is so small for today's systems that it does not exercise the NFS service. So we increased the size of the benchmark by a factor of n as follows: phase 1 and 2 create n copies of the source tree, and the other phases operate in all these copies. We ran a version of Andrew with n equal to 100, Andrew100, and another with n equal to 500, Andrew500. BFS builds a file system inside a memory mapped file [6]. We ran Andrew100 in a file system file with 205 MB and Andrew500 in a file system file with 1 GB; both benchmarks fill 90% of these files. Andrew100 fits in memory at both the client and the replicas but Andrew500 does not.

We also compare BFS and the NFS implementation in Linux, NFS-std. The performance of NFS-std is a good metric of what is acceptable because it is used daily by many users. For all configurations, the actual benchmark code ran at the client workstation using the standard NFS client implementation in the Linux kernel with the same mount options. The most relevant of these options for the benchmark are: UDP transport, 4096-byte read and write buffers, allowing write-back client caching, and allowing attribute caching.

Tables 2 and 3 present the results for these experiments. We report the mean of 3 runs of the benchmark. The standard deviation was always below 1% of the reported averages except for phase 1 where it was as high as 33%. The results show that BFS-PK takes 12 times longer than BFS to run Andrew100 and 15 times longer to run Andrew500. The slowdown is smaller than the one observed with the micro-benchmarks because the

phase	BFS-PK	BFS	NFS-std
1	25.4	0.7	0.6
2	1528.6	39.8	26.9
3	80.1	34.1	30.7
4	87.5	41.3	36.7
5	2935.1	265.4	237.1
total	4656.7	381.3	332.0

Table 2: Andrew100: elapsed time in seconds

client performs a significant amount of computation in this benchmark.

Both BFS and BFS-PK use the read-only optimization described in [6] for reads and lookups, and as a consequence do not set the time-last-accessed attribute when these operations are invoked. This reduces the performance difference between BFS and BFS-PK during phases 3 and 4 where most operations are read-only.

phase	BFS-PK	BFS	NFS-std
1	122.0	4.2	3.5
2	8080.4	204.5	139.6
3	387.5	170.2	157.4
4	496.0	262.8	232.7
5	23201.3	1561.2	1248.4
total	32287.2	2202.9	1781.6

Table 3: Andrew500: elapsed time in seconds

BFS-PK is impractical but BFS's performance is close to NFS-std: it performs only 15% slower in Andrew100 and 24% slower in Andrew500. The performance difference would be lower if Linux implemented NFS correctly. For example, we reported previously [6] that BFS was only 3% slower than NFS in Digital Unix, which implements the correct semantics. The NFS implementation in Linux does not ensure stability of modified data and meta-data as required by the NFS protocol, whereas BFS ensures stability through replication.

6.3 The Cost of Recovery

Frequent proactive recoveries and key changes improve resilience to faults by reducing the window of vulnerability, but they also degrade performance. We ran Andrew to determine the minimum window of vulnerability that can be achieved without overlapping recoveries. Then we configured the replicated file system to achieve this window, and measured the performance degradation relative to a system without recoveries.

The implementation of the proactive recovery mechanism is complete except that we are simulating the secure co-processor, the read-only memory, and the watchdog timer in software. We are also simulating fast reboots. The LinuxBIOS project [22] has been experimenting with replacing the BIOS with Linux. They claim to be able to reboot Linux in 35 s (0.1 s to get the kernel running and 34.9 to execute scripts in `/etc/rc.d`) [22]. This means that in a suitably configured machine we should be able to reboot in less than a second. Replicas simulate

a reboot by sleeping either 1 or 30 seconds and calling `msync` to invalidate the service-state pages (this forces reads from disk the next time they are accessed).

6.3.1 Recovery Time

The time to complete recovery determines the minimum window of vulnerability that can be achieved without overlaps. We measured the recovery time for Andrew100 and Andrew500 with 30s reboots and with the period between key changes, T_k , set to 15s.

Table 4 presents a breakdown of the maximum time to recover a replica in both benchmarks. Since the processes of checking the state for correctness and fetching missing updates over the network to bring the recovering replica up to date are executed in parallel, Table 4 presents a single line for both of them. The line labeled *restore state* only accounts for reading the log from disk the service state pages are read from disk on demand when they are checked.

	Andrew100	Andrew500
save state	2.84	6.3
reboot	30.05	30.05
restore state	0.09	0.30
estimation	0.21	0.15
send new-key	0.03	0.04
send request	0.03	0.03
fetch and check	9.34	106.81
total	42.59	143.68

Table 4: Andrew: recovery time in seconds.

The most significant components of the recovery time are the time to save the replica's log and service state to disk, the time to reboot, and the time to check and fetch state. The other components are insignificant. The time to reboot is the dominant component for Andrew100 and checking and fetching state account for most of the recovery time in Andrew500 because the state is bigger.

Given these times, we set the period between watchdog timeouts, T_w , to 3.5 minutes in Andrew100 and to 10 minutes in Andrew500. These settings correspond to a minimum window of vulnerability of 4 and 10.5 minutes, respectively. We also run the experiments for Andrew100 with a 1s reboot and the maximum time to complete recovery in this case was 13.3s. This enables a window of vulnerability of 1.5 minutes with T_w set to 1 minute.

Recovery must be fast to achieve a small window of vulnerability. While the current recovery times are low, it is possible to reduce them further. For example, the time to check the state can be reduced by periodically backing up the state onto a disk that is normally write-protected and by using copy-on-write to create copies of modified pages on a writable disk. This way only the modified pages need to be checked. If the read-only copy of the state is brought up to date frequently (e.g., daily), it will be possible to scale to very large states while achieving even lower recovery times.

6.3.2 Recovery Overhead

We also evaluated the impact of recovery on performance in the experimental setup described in the previous section. Table 5 shows the results. BFS-rec is BFS with proactive recoveries. The results show that adding frequent proactive recoveries to BFS has a low impact on performance: BFS-rec is 16% slower than BFS in Andrew100 and 2% slower in Andrew500. In Andrew100 with 1s reboot and a window of vulnerability of 1.5 minutes, the time to complete the benchmark was 482.4s; this is only 27% slower than the time without recoveries even though every 15s one replica starts a recovery.

The results also show that the period between key changes, T_k , can be small without impacting performance significantly. T_k could be smaller than 15s but it should be substantially larger than 3 message delays under normal load conditions to provide liveness.

system	Andrew100	Andrew500
BFS-rec	443.5	2257.8
BFS	381.3	2202.9
NFS-std	332.0	1781.6

Table 5: Andrew: recovery overhead in seconds.

There are several reasons why recoveries have a low impact on performance. The most obvious is that recoveries are staggered such that there is never more than one replica recovering; this allows the remaining replicas to continue processing client requests. But it is necessary to perform a view change whenever recovery is applied to the current primary and the clients cannot obtain further service until the view change completes. These view changes are inexpensive because a primary multicasts a view-change message just before its recovery starts and this causes the other replicas to move to the next view immediately.

7 Related Work

Most previous work on replication techniques assumed benign faults, e.g., [17, 23, 18, 19] or a synchronous system model, e.g., [28]. Earlier Byzantine-fault-tolerant systems [26, 16, 20], including the algorithm we described in [6], could guarantee safety only if fewer than 1/3 of the replicas were faulty during the lifetime of the system. This guarantee is too weak for long-lived systems. Our system improves this guarantee by recovering replicas proactively and frequently; it can tolerate any number of faults if fewer than 1/3 of the replicas become faulty within a window of vulnerability, which can be made small under normal load conditions with low impact on performance.

In a previous paper [6], we described a system that tolerated Byzantine faults in asynchronous systems and performed well. This paper extends that work by providing recovery, a state transfer mechanism, and a new view change mechanism that enables both recovery and an important optimization — the use of MACs instead of public-key cryptography.

Rampart [26] and SecureRing [16] provide group membership protocols that can be used to implement recovery, but only in the presence of benign faults. These approaches cannot be guaranteed to work in the presence of Byzantine faults for two reasons. First, the system may be unable to provide safety if a replica that is not faulty is removed from the group to be recovered. Second, the algorithms rely on messages signed by replicas even after they are removed from the group and there is no way to prevent attackers from impersonating removed replicas that they controlled.

The problem of efficient state transfer has not been addressed by previous work on Byzantine-fault-tolerant replication. We present an efficient state transfer mechanism that enables frequent proactive recoveries with low performance degradation.

Public-key cryptography was the major performance bottleneck in previous systems [26, 16] despite the fact that these systems include sophisticated techniques to reduce the cost of public-key cryptography at the expense of security or latency. They cannot use MACs instead of signatures because they rely on the extra power of digital signatures to work correctly: signatures allow the receiver of a message to prove to others that the message is authentic, whereas this may be impossible with MACs. The view change mechanism described in this paper does not require signatures. It allows public-key cryptography to be eliminated, except for obtaining new secret keys. This approach improves performance by up to two orders of magnitude without losing security.

The concept of a system that can tolerate more than f faults provided no more than f nodes in the system become faulty in some time window was introduced in [24]. This concept has previously been applied in synchronous systems to secret-sharing schemes [13], threshold cryptography [14], and more recently secure information storage and retrieval [10] (which provides single-writer single-reader replicated variables). But our algorithm is more general; it allows a group of nodes in an asynchronous system to implement an arbitrary state machine.

8 Conclusions

This paper has described a new state-machine replication system that offers both integrity and high availability in the presence of Byzantine faults. The new system can be used to implement real services because it performs well, works in asynchronous systems like the Internet, and recovers replicas to enable long-lived services.

The system described here improves the security and robustness against software errors of previous systems by recovering replicas proactively and frequently. It can tolerate any number of faults provided fewer than 1/3 of the replicas become faulty within a window of vulnerability. This window can be small (e.g., a few minutes) under normal load conditions and when the attacker does not corrupt replicas' copies of the service state. Additionally, our system provides *intrusion*

detection; it detects denial-of-service attacks aimed at increasing the window and detects the corruption of the state of a recovering replica.

Recovery from Byzantine faults is harder than recovery from benign faults for several reasons: the recovery protocol itself needs to tolerate other Byzantine-faulty replicas; replicas must be recovered proactively; and attackers must be prevented from impersonating recovered replicas that they controlled. For example, the last requirement prevents signatures in messages from being valid indefinitely. However, this leads to a further problem, since replicas may be unable to prove to a third party that some message they received is authentic (because its signature is no longer valid). All previous state-machine replication algorithms relied on such proofs. Our algorithm does not rely on these proofs and has the added advantage of enabling the use of symmetric cryptography for authentication of all protocol messages. This eliminates the use of public-key cryptography, the major performance bottleneck in previous systems.

The algorithm has been implemented as a generic program library with a simple interface that can be used to provide Byzantine-fault-tolerant versions of different services. We used the library to implement BFS, a replicated NFS service, and ran experiments to determine the performance impact of our techniques by comparing BFS with an unreplicated NFS. The experiments show that it is possible to use our algorithm to implement real services with performance close to that of an unreplicated service. Furthermore, they show that the window of vulnerability can be made very small: 1.5 to 10 minutes with only 2% to 27% degradation in performance.

Acknowledgments

We would like to thank Kyle Jamieson, Rodrigo Rodrigues, Bill Weihl, and the anonymous referees for their helpful comments on drafts of this paper. We also thank the Computer Resource Services staff in our laboratory for lending us a switch to run the experiments and Ted Krovetz for the UMAC code.

References

- [1] M. Bellare and D. Micciancio. A New Paradigm for Collision-free Hashing: Incrementality at Reduced Cost. In *Advances in Cryptology - EUROCRYPT*, 1997.
- [2] J. Black et al. UMAC: Fast and Secure Message Authentication. In *Advances in Cryptology - CRYPTO*, 1999.
- [3] R. Canetti, S. Halevi, and A. Herzberg. Maintaining Authenticated Communication in the Presence of Break-ins. In *ACM Conference on Computers and Communication Security*, 1997.
- [4] M. Castro. *Practical Byzantine Fault Tolerance*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 2000. In preparation.
- [5] M. Castro and B. Liskov. A Correctness Proof for a Practical Byzantine-Fault-Tolerant Replication Algorithm. Technical Memo MIT/LCS/TM-590, MIT Laboratory for Computer Science, 1999.
- [6] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *USENIX Symposium on Operating Systems Design and Implementation*, 1999.
- [7] C. Collberg and C. Thomborson. Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection. Technical Report 2000-03, University of Arizona, 2000.
- [8] S. Floyd et al. A Reliable Multicast Framework for Lightweight Sessions and Application Level Framing. *IEEE/ACM Transactions on Networking*, 5(6), 1995.
- [9] S. Forrest et al. Building Diverse Computer Systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, 1997.
- [10] J. Garay et al. Secure Distributed Storage and Retrieval. *Theoretical Computer Science*, to appear.
- [11] L. Gong. A Security Risk of Depending on Synchronized Clocks. *Operating Systems Review*, 26(1):49-53, 1992.
- [12] M. Herlihy and J. Wing. Axioms for Concurrent Objects. In *ACM Symposium on Principles of Programming Languages*, 1987.
- [13] A. Herzberg et al. Proactive Secret Sharing, Or: How To Cope With Perpetual Leakage. In *Advances in Cryptology - CRYPTO*, 1995.
- [14] A. Herzberg et al. Proactive Public Key and Signature Systems. In *ACM Conference on Computers and Communication Security*, 1997.
- [15] J. Howard et al. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1), 1988.
- [16] K. Kihlstrom, L. Moser, and P. Melliar-Smith. The SecureRing Protocols for Securing Group Communication. In *Hawaii International Conference on System Sciences*, 1998.
- [17] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7), 1978.
- [18] L. Lamport. The Part-Time Parliament. Technical Report 49, DEC Systems Research Center, 1989.
- [19] B. Liskov et al. Replication in the Harp File System. In *ACM Symposium on Operating System Principles*, 1991.
- [20] D. Malkhi and M. Reiter. Secure and Scalable Replication in Phalanx. In *IEEE Symposium on Reliable Distributed Systems*, 1998.
- [21] D. Mazières et al. Separating Key Management from File System Security. In *ACM Symposium on Operating System Principles*, 1999.
- [22] Ron Minnich. The Linux BIOS Home Page. <http://www.acl.lanl.gov/linuxbios>, 2000.
- [23] B. Oki and B. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *ACM Symposium on Principles of Distributed Computing*, 1988.
- [24] R. Ostrovsky and M. Yung. How to Withstand Mobile Virus Attacks. In *ACM Symposium on Principles of Distributed Computing*, 1991.
- [25] J. Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? In *USENIX Summer*, 1990.
- [26] M. Reiter. The Rampart Toolkit for Building High-Integrity Services. *Theory and Practice in Distributed Systems (LNCS 938)*, 1995.
- [27] R. Rivest. The MD5 Message-Digest Algorithm. Internet RFC-1321, 1992.
- [28] F. Schneider. Implementing Fault-Tolerant Services Using The State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4), 1990.

Exploring Failure Transparency and the Limits of Generic Recovery

David E. Lowell

Western Research Laboratory
Compaq Computer Corporation
david.lowell@compaq.com

Subhachandra Chandra Peter M. Chen

Department of Electrical Engineering and Computer Science
University of Michigan
{schandra, pmchen}@eecs.umich.edu

Abstract: We explore the abstraction of failure transparency in which the operating system provides the illusion of failure-free operation. To provide failure transparency, an operating system must recover applications after hardware, operating system, and application failures, and must do so without help from the programmer or unduly slowing failure-free performance. We describe two invariants that must be upheld to provide failure transparency: one that ensures sufficient application state is saved to guarantee the user cannot discern failures, and another that ensures sufficient application state is lost to allow recovery from failures affecting application state. We find that several real applications get failure transparency in the presence of simple stop failures with overhead of 0-12%. Less encouragingly, we find that applications violate one invariant in the course of upholding the other for more than 90% of application faults and 3-15% of operating system faults, rendering transparent recovery impossible for these cases.

1. Introduction

One of the most important jobs of the operating system is to conceal the complexities and inadequacies of the underlying machine. Towards this end, modern operating systems provide a variety of abstractions. To conceal machines' limited memory, for example, operating systems provide the abstraction of practically boundless virtual memory. Similarly, operating systems give the abstraction of multithreading for those applications that might benefit from more processors than are present in hardware.

Failures by computer system components, be they hardware, software, or the application, are a shortcoming of modern systems that has not been abstracted away. Instead, computer programmers and users routinely have to deal with the effects of failures, even on machines running state-of-the-art operating systems.

With this paper we explore the abstraction of *failure transparency* in which the operating system generates the illusion of failure-free operation. To provide this illusion, the operating system must handle all hardware, software, and application failures to keep them from affecting what the user sees. Furthermore, the operating system must do so without help from the programmer and without unduly slowing down failure-free operation.

Fault-tolerance research has established many of the components of failure transparency, such as programmer-transparent recovery [4, 11, 25, 28], and recovery for general applications [4, 14]. Some researchers have even discussed handling application failures [13, 17, 31].

However, significant questions surrounding failure transparency remain. The focus of this paper is on delving into several of these unanswered questions. First, we will explore the question "how does one guarantee failure transparency in general?" The answer to this question comes in the form of two invariants. The first invariant is a reformulation of existing recovery theory, governing when an application must save its work to ensure that the user does not discern failures. In contrast, the second invariant governs how much work an application must lose to avoid forcing the same failure during recovery.

The Save-work invariant can require applications to commit their state frequently to stable storage. The question therefore arises "how expensive is it for general applications to uphold the Save-work invariant?" In answering this question we find, to our surprise, that even complex, general applications are able to efficiently uphold Save-work.

Given that the Save-work invariant forces applications to preserve work and the Lose-work invariant forces applications to throw work away, we conclude by investigating the question, "how often do these invariants conflict, making failure transparency impossible?" The unfortunate answer is that the invariants conflict all too often.

2. Guaranteeing Failure Transparency

We first delve into the question: *how does one guarantee failure transparency in general?* Our exploration begins with a synthesis of existing recovery theory that culminates in the Save-work invariant. In Section 2.4, we then extend recovery theory to point out a parameterization of the space of recovery protocols, as well as the relationship between protocols at different points in the space. Finally, we develop a new theory and second invariant for ensuring the possibility of recovery from failures that affect application state.

2.1. Primitives for general recovery

In attempting to provide failure transparency, the goal is to recover applications using only general techniques that

require no help from the application. There are several recovery primitives available to us in this domain: *commit events*, *rollback* of a process, and *reexecution* from a prior state.

A process can execute commit events to aid recovery after a failure. By executing a commit event, a process preserves its state at the time of the commit so that it can later restore that state and continue execution. Although how commit events are implemented is not important to our discussion, executing a commit event might involve writing out a full-process checkpoint to stable storage, ending a transaction, or sending a state-update message to a backup process.

When a failure occurs, the application undergoes rollback of its failed processes; each failed process is returned to its last committed state. From that state, the recovering process begins reexecution, possibly recomputing work lost in the failure.

Providing generic recovery requires that applications tolerate forced rollback and reexecution. As a result, all application operations must be either undoable or redoable.

Most application operations that simply modify process state are easily undone. However, some events, such as message sends, are hard to undo. Undoing a send can involve the added challenge of rolling back the recipient's state. Other events can be impossible to undo. For example, we cannot undo the effects on the user resulting from visible output. However, systems providing failure transparency ensure that these user-visible events will never be undone.

Similarly, since simple state changes by the application are idempotent, most application events can be safely redone. However, events like message sends and receives are more difficult to redo. For message send events to be redoable, the application must either tolerate or filter duplicate messages. For receive events to be redoable, messages must be saved at either the sender or receiver so they can be re-delivered after a failure. Luckily, these reexecution requirements are very similar to the demands made of systems that transmit messages on unreliable channels (e.g. UDP). Such systems must already work correctly even with lost or duplicated messages. For many recovery systems, an application or protocol layer's natural filtering and retransmission mechanisms will be enough to support the needs of reexecution recovery. For others, messages may have to be held in a recovery buffer of some kind so they can be re-delivered should a receive event be redone.

2.2. Computation and failure model

We will informally present a recovery theory that will let us relate the challenge of guaranteeing failure transparency to the precise events executed by an application. For a more formal version of the theory, please see [22].

We begin by constructing a model of computing. One or more processes working together on a task is called a *computation*. We model each process as a finite state

machine. That is, each process has state and computes by transitioning from state to state according to the inputs it receives. Each state transition executed by a process is an *event*. An event e_p^i is the i 'th event executed by process p . Events can correspond in real programs to simple changes of application state, sending and receiving messages, and so on. We call events that have an effect on the user *visible events* (these events have traditionally been called "output events" [11]). Under our model, computation proceeds *asynchronously*, that is, without known bounds on message delivery time or the relative speeds of processes.

As needed, we will order events in our asynchronous computations with Lamport's *happens-before* relation [19]. We may also need to discuss the causal relationship between events. For example, we may need to ask, "did event e in some way *cause* event e' ?" We will use *happens-before* as an approximation of causality. We will however distinguish between *happens-before*'s use as an ordering constraint and its use as an approximation of causality by using the expression *causally precedes* in this latter role. That is, we say event e *causally precedes* event e' if and only if e *happens-before* e' and we intend to convey that e causes event e' .

We will consider failures of two forms. A *stop failure* is one in which execution of one or more processes in the computation simply ceases. Stop failures do occur in real systems—the loss of power, the frying of a processor, or the abrupt halting of the operating system all appear to the recovery system as stop failures. Since stop failures instantaneously stop the execution of the application and do not corrupt application state, recovering from them is relatively easy.

Harder to handle are *propagation failures*. We define a propagation failure to be one in which a bug somewhere in the system causes the application to enter a state it would not enter in a failure-free execution. A propagation failure can begin with a bug in hardware, the operating system, or the application. Bugs in the application are always propagation failures, but bugs in hardware and the operating system are propagation failures only once they affect application state.

Recovering from propagation failures is hard because a process can execute for some time after the failure is triggered. During that time the process can propagate buggy data into larger portions of its state, to other processes, or onto stable storage.

We can imagine bugs that do not cause crashes, but that simply cause incorrect visible output by the application. However, our focus with this work is on recovering from failures. Therefore, we will assume that applications will detect faults and fail before generating incorrect output.

2.3. Failure transparency for stop failures

We start by examining how to ensure failure transparency in the presence of stop failures. We must first fix a precise notion of "correct" recovery from failures. We could

establish almost any standard: recovering the exact pre-failure state, losing less than 10 seconds of work, and so on. However, given that our end goal is to mask failures from the user, we will define correct recovery in terms of the application output seen by the user.

Given a computation in which processes have failed, recovered, and continued execution:

Definition: Consistent Recovery

Recovery is *consistent* if and only if there exists a complete, failure-free execution of the computation that would result in a sequence of visible events equivalent to the sequence of visible events actually output in the failed and recovered run.

Thus for an application's recovery to be consistent, the sum total of the application's visible output before and after a failure must be equivalent to the output from some failure-free execution of the application.

It is possible that many different modes of consistent recovery could be allowed depending on how one defines "equivalent". For our purposes, we will call a sequence of visible events V output by a recovered computation equivalent to sequence V' output by a failure-free run if the only events in V that differ from V' are repeats of earlier events from V .

We use equivalence in which duplicate visible events are allowed because guaranteeing no duplication is very hard (exactly once delivery problem). Furthermore, allowing duplicates provides some flexibility in how one attains consistent recovery. More importantly, users can probably overlook duplicated visible events. See [22] for a more detailed discussion of equivalence.

Our definition of consistent recovery places two constraints on recovering applications. First, computations must always execute visible events that extend a legal, failure-free sequence of visible events, even in the presence of failures. We will call this the *visible constraint*. Second, computations must always be able to execute to completion. This latter constraint follows from the fact that consistent recovery is defined in terms of *complete* sequences of visible events. If a failure prevents an application from running to completion, its sequence can never be complete. For reasons that will become clear later, we will call this second constraint on recovery the *no-orphan constraint*.

Although consistent recovery and failure transparency are closely related, they are not the same thing. Providing failure transparency amounts to guaranteeing consistent recovery without any help from the application, and without slowing the application's execution appreciably.

Our next task is to examine how to guarantee applications get consistent recovery. One particular class of events poses the greatest challenge: *non-deterministic events*. In a state-machine, a non-deterministic event is a transition from a state that has multiple possible next states. For example, in

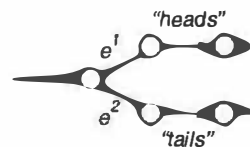


Figure 1: Coin-flip application. Depending on whether non-deterministic event e^1 or e^2 gets executed, the application executes one of two possible visible events.

Figure 1, events e^1 and e^2 are both non-deterministic. In real systems, non-deterministic events correspond to actions that can have different results before and after a failure, like checking the time-of-day clock, taking a signal, reading user input, or receiving a message.

Non-deterministic events are intimately related to consistent recovery. To see how, again consider the application shown in Figure 1. Imagine that the application executes non-deterministic event e^1 , then the visible event "heads", then fails. Then during recovery imagine that the application rolls back and this time executes e^2 followed by the visible event "tails". Although this application can correctly output either *heads* or *tails*, in no correct execution does it output both *heads* and *tails*. Therefore, recovery in this example is not consistent and our sample application's non-deterministic events are the culprits.

As discussed in Section 2.1, applications can execute commit events to aid later rollback. We would like to use commit events to guarantee consistent recovery, avoiding the inconsistency non-deterministic events can cause. The following theorem provides the necessary and sufficient condition for doing exactly that under stop failures.

Save-work Theorem

A computation is guaranteed consistent recovery from stop failures if and only if for each executed non-deterministic event e_p^i that *causally precedes* a visible or commit event e , process p executes a commit event e_p^j such that e_p^j *happens-before* (or atomic with) e , and $i \leq j$.

This theorem dictates when processes must commit in order to ensure consistent recovery. At the heart of this theorem is the Save-work invariant, which informally states "each process has to commit all its non-deterministic events that causally precede visible or commit events". We can further divide this invariant into separate rules, one that enforces the visible constraint of consistent recovery, and one that enforces the no-orphan constraint. If we follow the rule "commit every non-deterministic event that causally precedes a visible event", we are assured that the application's visible output will always extend a legal sequence of visible events. We'll call this the Save-work-visible invariant. If we follow the rule "commit every non-deterministic event that causally precedes a commit event", we are assured that a finite number of stop failures cannot prevent the application from executing to completion. We'll call this the

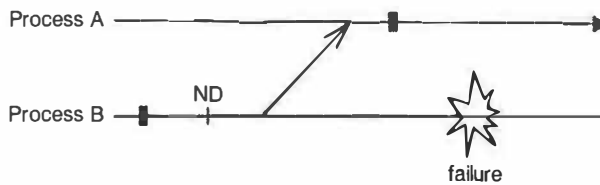


Figure 2: A problematic distributed computation. We see two processes' timelines. The arrow between the processes represents a message from B to A. Black boxes represent commits. The event marked "ND" is a non-deterministic event. Process A is an orphan after Process B's failure as A has committed a dependence on B's lost non-deterministic event.

Save-work-orphan invariant. To better understand this latter rule, consider the computation depicted in Figure 2.

A process is called an *orphan* if it has committed a dependence on another process's non-deterministic event that has been lost and may not be reexecuted. For example, Process A in Figure 2 is an orphan because it has committed its dependence on Process B's lost non-deterministic event.

An orphan can prevent an application from executing to completion when it is upholding Save-work-visible. Consider an orphan that has committed a dependence on a lost non-deterministic event e_p^{ND} . If the orphan attempts to execute a visible event e , Save-work-visible requires that process p commit e_p^{ND} . However, since process p has already failed and aborted e_p^{ND} , it cannot commit it. Furthermore, since the orphan cannot abort its dependence on e_p^{ND} , it can never execute e and the computation will not be able to complete.

The remedy for this scenario is to uphold Save-work-orphan, which ensures that any non-deterministic event that causally precedes a commit is committed.

We must make two assumptions for the Save-work Theorem to be necessary. We ensure the necessity of Save-work-visible by assuming that all non-deterministic events can cause inconsistency. We ensure the necessity of Save-work-orphan by assuming that all processes in the computation affect the computation's visible output. For the details of these assumptions as well as the proof of the Save-work Theorem, please see [22].

2.4. Upholding Save-work

There are many ways an application can uphold the Save-work invariant to ensure consistent recovery for stop failures. For example, an application can execute a commit event for every event executed by the application. Although such a protocol will cause a very large number of commits, it has the advantage of being trivial to implement: the protocol does not need to figure out which events are non-deterministic, or which events are visible. Even without knowing event types, it correctly upholds the Save-work invariant.

Consider a protocol in which each process executes a commit event immediately after each non-deterministic event. In committing all non-deterministic events, this proto-

col will certainly commit those non-deterministic events that causally precede visible or commit events. Therefore it upholds Save-work and will guarantee consistent recovery. We call this protocol Commit After Non-Deterministic, or CAND.

We can also uphold Save-work without knowing about the non-determinism in the computation. Under the Commit Prior to Visible or Send protocol (CPVS), each process commits just before doing a visible event or a send to another process. When a process commits before each of its visible events, it is assured that all its non-determinism that causally precedes the visible event is committed. If each process also commits before every send event, then it cannot pass a dependence on an uncommitted non-deterministic event to another process. Thus, CPVS also upholds Save-work.

The Commit Between Non-Deterministic and Visible or Send (CBNDVS) protocol takes advantage of knowledge of both non-determinism and visible and send events in order to uphold Save-work. Under this protocol, each process commits immediately before a visible or send event if the process has executed a non-deterministic event since its last commit.

Since commit events can involve writing lots of data to stable storage, they can be slow. Therefore, minimizing the number of commits executed can be important to failure-free performance. There exist several general techniques for minimizing commits.

Logging is a general technique for reducing an application's non-determinism [12]. If an application writes the result of a non-deterministic event to a persistent log, and then uses that log record during recovery to ensure the event executes with the same result, the event is effectively rendered deterministic. Logging some of an application's non-determinism can significantly reduce commit frequency. Logging *all* an application's non-determinism lets the application uphold Save-work without committing at all.

Tracking whether one process's non-determinism causally precedes events on another process can be complex. In fact, we can think of the CPVS protocol as pessimistically committing before send events rather than track causality between processes. However, applications can avoid committing before sends without tracking causality by employing a distributed commit, such as *two-phase commit* (2PC)—all processes would commit whenever any process does a visible event. Using two-phase commit can reduce commit frequency if visible events are less frequent than sends. Applications can further reduce commits by tracking causality between processes, involving in the coordinated commit only those processes with relevant non-deterministic events.

Not only can each of these protocols be viewed as a different technique for upholding Save-work, but so can all existing protocols from the recovery literature.

For example, pure message logging protocols make all message receive events deterministic, allowing applications

whose only non-deterministic events are receives to uphold Save-work without committing. The different message logging protocols differ in how the logging is carried out. For example, *Sender-based Logging* (SBL) protocols keep the log record for the receive event in the volatile memory of the sender [15], while *Family-based Logging* (FBL) keeps log entries in the memory of downstream processes [2].

In the *Manetho* system, each process maintains log records for all the non-deterministic events it depends on in an *antecedence graph*. When a process wants to execute a visible event, it upholds Save-work by writing the antecedence graph to stable storage [11]. In the *Optimistic Logging* protocol, processes write log records to stable storage asynchronously [28]. When a process wants to do a visible event, it upholds Save-work by first waiting for all relevant log records to make it to disk.

The *Targon/32* system attempts to handle more non-determinism than these other logging protocols [4]. All sources of non-determinism except signals are converted into messages that are logged in the memory of a backup process on another processor. Whenever a signal is delivered (an event that remains non-deterministic), Targon/32 forces a commit to uphold Save-work. The *Hypervisor* system logs *all* sources of non-determinism using a virtual machine under the operating system [5].

Under a Coordinated Checkpointing protocol, a process executing a visible event essentially assumes that all processes in the computation with which it has recently communicated have executed non-deterministic events that causally precede the visible event [18]. To uphold the Save-work invariant, the process executing the visible event initiates an agreement protocol to force all these other processes to commit.

Each of these recovery protocols represents a different technique for upholding Save-work. Each to varying degrees trades off programmer effort and system complexity for reduced commit frequency (and hopefully overhead).

Some protocols focus their effort to reduce commit frequency on the challenge of identifying and reducing non-determinism. Others endeavor to use knowledge of an application's visible events. Still others do some of each. Each protocol can be seen as representing a point in a two-dimensional space of protocols. One axis in the space represents effort made to identify and possibly convert application non-determinism. The other axis represents effort made to identify visible events and to commit as few non-visible events as possible.

Such a protocol space is useful because it helps us understand the relationships between historically disparate protocols and to identify new ones. Figure 3 shows how the protocols we have described in this section might appear in such a protocol space.

A protocol falling at the origin of the space would uphold Save-work by committing every event executed by

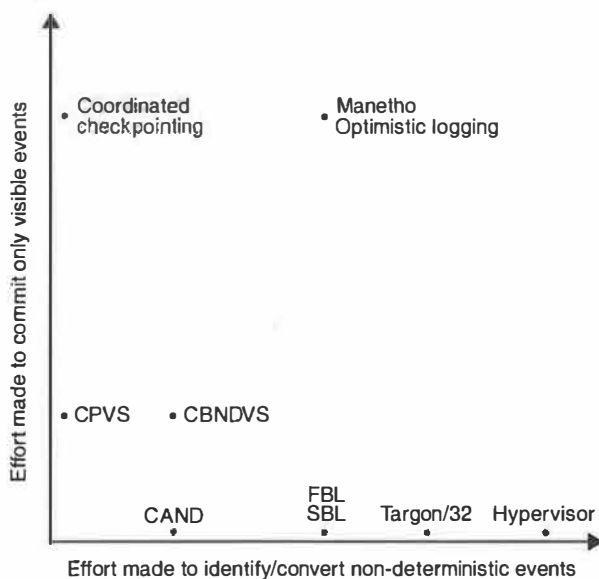


figure 3: Protocol space. All consistent recovery protocols fall somewhere in this space. Some protocols focus on dealing with on-determinism, while others concern themselves with visible vents. Some do a little of each.

each process, exerting no effort to determine which events are non-deterministic or visible. As protocols fall further out the horizontal axis, they make sufficient effort to recognize that some events are deterministic and therefore do not require commits. At the point occupied by CAND, the protocol makes sufficient effort to distinguish all of the application's deterministic and non-deterministic events, executing a commit only after non-deterministic ones. Beyond that point, the protocols begin to employ logging, exerting effort to convert more and more of the application's non-deterministic events into deterministic ones. A protocol in that portion of the space forces a commit only when the application executes some unlogged non-determinism. At the point occupied by Hypervisor, the protocol makes sufficient effort to log all non-determinism, never forcing a commit.

For the vertical axis, we can think of the protocol at the origin as committing all events rather than exert the effort needed to determine which events are visible. Protocols falling further up the axis exert more effort to avoid committing events that are not visible. At the point occupied by CPVS, protocols commit only the true visible events and send events—committing before sends takes less effort than tracking whether that send leads to a visible event on another process. Protocols falling yet further up in the space (such as Coordinated Checkpointing) are able to ask remote processes to commit if needed. Under those protocols, applications are forced to commit before visible events only.

Some protocols fall in the middle of the space, applying techniques both for identifying and converting non-determinism, as well as for tracking the causal relationship

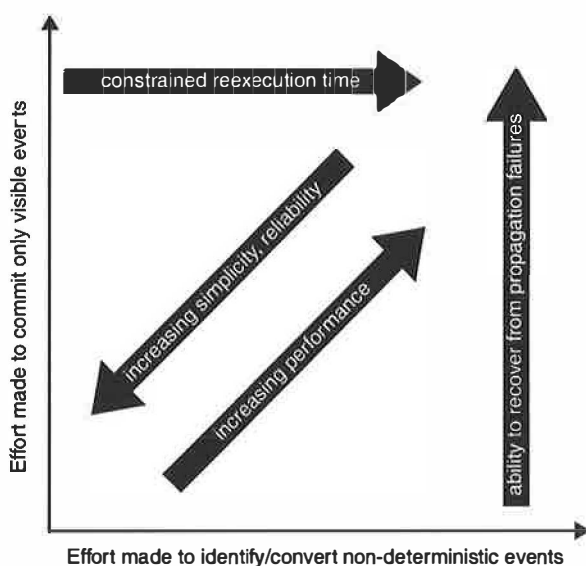


Figure 4: Protocol space with other design variables.

between non-deterministic events and the visible events they cause.

Although all protocols in the space are equivalent in terms of upholding Save-work, they do differ in terms of other design variables. As shown in Figure 4, we can map trends in several important design variables onto the protocol space.

The farther a protocol falls from the origin, the lower its commit frequency is likely to be, and therefore, the better its performance. However, this improved performance comes at the expense of simplicity and reliability. Protocols close to the origin are very simple to implement, and therefore are more likely to be implemented correctly.

For protocols that fall on the vertical axis, the recovery system needs only rollback failed processes and let them continue normally. Protocols further to the right in the protocol space have longer recovery times because after rollback, the recovery system must for some time constrain reexecution to follow the path taken before the failure.

The further a protocol falls from the horizontal axis, the more non-determinism it safely leaves in the application. As we will discuss in Section 2.6, the more non-determinism in an application, the better the chance it will survive propagation failures.

2.5. Failure transparency for stop and propagation failures

As mentioned in Section 2.2, failures can take two forms: stop failures and propagation failures. Upholding the Save-work invariant is enough to guarantee consistent recovery only in the presence of stop failures. To illustrate this observation, consider a protocol that commits all events a process executes. This protocol clearly upholds Save-work. However, if the process experiences a propagation failure

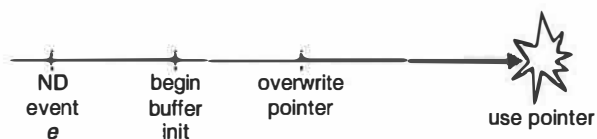


figure 5: Sample propagation failure timeline. A non-deterministic event e causes buffer initialization to overflow and trash a pointer. A commit any time after e will prevent recovery from this failure.

(which by definition involves executing buggy events), this protocol is guaranteed to commit buggy state. As a result, the process will fail again during recovery, and the application will never be able to complete after the failure.

Thus, in order to guarantee consistent recovery in the presence of propagation failures, an application must not only commit to uphold Save-work, but when it commits it must avoid preserving the conditions of its failure. In this section we examine what exactly an application must do to guarantee consistent recovery in the presence of propagation failures.

As was the case in our discussion of consistent recovery, non-deterministic events are central to the issue of recovering from propagation failures. Imagine an application that, as a result of non-deterministic event e , overruns a buffer it is clearing and zeroes out a pointer down the stack (see Figure 5). Later, it attempts to dereference the pointer and crashes. Obviously if the application commits after zeroing the pointer, recovery is doomed. However, if the application commits any time *before* zeroing the pointer and after e , recovery will still be doomed if there are no other non-deterministic events after e . In this case, the pointer is not corrupted in the last committed state, but it is guaranteed to be *re-corrupted* during recovery.

Note that had the application committed just before e and not after, all could be well. During recovery, the application would redo the non-deterministic event which could execute with a different result and avoid this failure altogether.

Thus non-determinism helps our prospects for recovering from propagation failures by limiting the scope of what is preserved by a commit.

But, not all non-determinism is created equal in this regard. In building up the Save-work invariant, we conservatively treated as non-deterministic any event that could conceivably have a different result during recovery. However, some non-deterministic events are likely to have the same result before and after a failure, and the recovery system cannot depend on these events to change after recovery. We will call these events *fixed non-deterministic* events.

A common example of a fixed non-deterministic event is user input. We cannot depend on the user to aid recovery by entering different input values after a failure. Other examples of fixed non-deterministic events include non-deterministic events whose results are based on the fullness of the

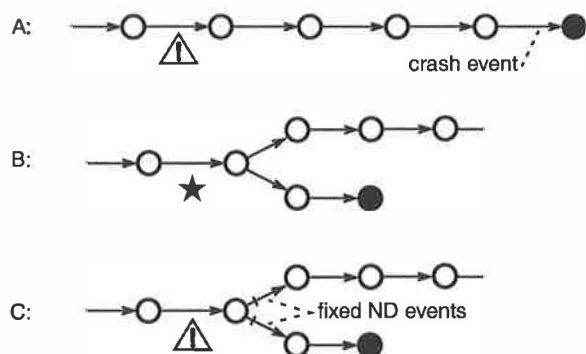


Figure 6: Three sample machines with crash events (events that end states filled black). It is okay to commit in case B at the point marked. Committing either A or C where marked could prevent recovery.

disk (such as the `write` system call), or that depend on the number of slots left in the operating system's open file table (such as the `open` system call).

Non-deterministic events that are not fixed we will call *transient non-deterministic* events. Scheduler decisions, signals, message ordering, the timing of user input, and system calls like `gettimeofday` are all transient non-deterministic events.

We need to incorporate into our computational model a way to represent the eventual crash of a process during a propagation failure. We will model a process's crash as the execution of a *crash event*. When a process executes a crash event, it transitions into a state from which it cannot continue execution. In the example in Figure 5, the crash event is the dereferencing of the null pointer.

As mentioned above, an untimely commit during a propagation failure can ensure that recovery fails. Let us examine in more detail when a process should not commit.

Clearly a process should not commit while executing a string of deterministic events that end in a crash event. Doing so is guaranteed to either commit the buggy state that leads to the crash, or to ensure that the faulty state is regenerated during recovery. This case is shown in Figure 6A.

However, a process can safely commit before a transient non-deterministic event as long as at least one of the possible results of that event does not lead to the execution of a crash event (see Figure 6B).

How about committing before a fixed non-deterministic event where one of the event's possible results leads to a crash? This case is shown in Figure 6C. If the application commits before the fixed non-deterministic event, recovery is possible only if the event executes with a result that leads down the path not including the crash event. If the application is unlucky and the fixed non-deterministic event sends the application down the path towards the crash, the commit will ensure recovery always fails. Since we cannot rely on

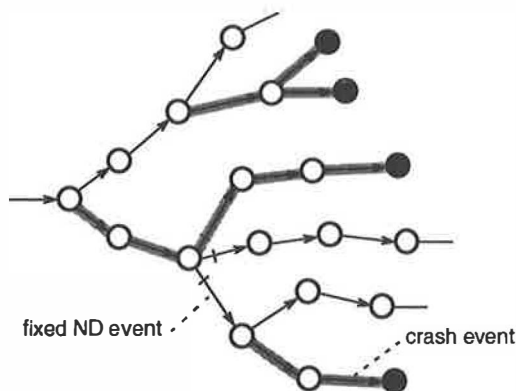


Figure 7: Portion of a state machine, its crash events, and corresponding dangerous paths. Crash events are those that end in states filled black. Fixed non-deterministic events are marked with a slash. The shaded paths are dangerous.

fixed non-deterministic events having results conducive to recovery, we cannot commit before any fixed non-deterministic events that might lead to a crash.

We can infer that some paths through a portion of a state machine are problematic for handling propagation failures—committing anywhere along the paths could prevent recovery. We next present an algorithm for finding these paths. For this discussion, we assume perfect knowledge of each process's crash events. We recognize that this is not practical—if we knew all the crash events, we could likely fix all the bugs! However, making this assumption will help us to analyze when recovery is possible with the best possible knowledge.

Given a single process's state machine and its crash events:

Single-Process Dangerous Paths Algorithm

- Color all crash events in the state machine.
- Color an event e if all events out of e 's end state are colored.
- Color an event e if at least one event out of e 's end state is colored and is a fixed non-deterministic event.

We call all the paths in the state machine colored by this algorithm *dangerous paths*. A portion of a state machine with its dangerous paths highlighted is shown in Figure 7.

We now present without proof a theorem which governs when recovery is possible in the presence of propagation failures.

Lose-work Theorem

Application-generic recovery from propagation failures is guaranteed to be possible if and only if the application executes no commit event on a dangerous path.

This theorem provides an invariant for ensuring the possibility of recovery from propagation failures: processes must not commit on dangerous paths. It is interesting to note that the location of the initial bug that caused the crash is surprisingly irrelevant. In the end, all that matters is the eventual crash event (or events) that result from that bug and its location relative to the application's transient non-deterministic events.

How about for multi-process applications? The challenge for distributed applications is in computing their dangerous paths. Unlike the dangerous paths algorithm presented above, computing dangerous paths for a distributed application cannot be done statically: whether one process's path is dangerous can depend on the paths taken by the other processes in the computation and where they have committed.

Given a process *P* that wants to determine its dangerous paths (presumably so it can commit without violating Lose-work):

Multi-Process Dangerous Paths Algorithm

- Process *P* collects a snapshot of when each process in the computation last committed.
- For each non-deterministic receive event that *P* has executed, treat that receive as a transient non-deterministic event if the sender's last commit occurred before the send, and the sender executed a transient non-deterministic event between its last commit and the send. All other receives *P* has executed are fixed non-deterministic events.
- Run the single-process dangerous paths algorithm to compute *P*'s dangerous paths.

2.6. Upholding Lose-work

The simplest way to uphold Lose-work is to ensure that no process ever commits. Although this solution has the advantage of requiring no application-specific knowledge to implement, it also prohibits guaranteeing consistent recovery.

Clearly, without perfect knowledge of the application's non-determinism and crash events it is impossible to guarantee a committing application upholds Lose-work. Despite the impossibility of directly upholding the invariant, we can use the Lose-work Theorem to draw some conclusions about recovering from propagation failures.

First, we observe that it is impossible to uphold both Save-work and Lose-work for some applications. Consider an application with a visible event on a dangerous path. The dangerous path will extend back at least to the last non-deterministic event. Upholding Save-work forces the application to commit between the last non-deterministic event and the visible event, which will violate Lose-work.

Second, some protocols designed to uphold Save-work for stop failures *guarantee* that applications will not recover

from propagation failures. These protocols either commit or convert all non-determinism, ensuring a commit after the non-deterministic event that steers a process onto a dangerous path, thus violating Lose-work. CAND, Sender-based logging, Targon/32, and Hypervisor are all examples of protocols that prevent applications from surviving propagation failures. Indeed, any protocol that falls on the horizontal axis of the Save-work protocol space (see Figure 3) will prevent upholding Lose-work. The farther a protocol falls from the horizontal axis, the more it focuses its attention on handling visible events and the more non-determinism it leaves safely uncommitted, thus decreasing the chances of violating Lose-work (see Figure 4).

Although directly upholding Lose-work is impossible, some applications with mostly "non-repeatable" bugs (so called "Heisenbugs" [13]) may be able to commit with a low probability of violating the invariant. There are also a number of ways applications can deliberately endeavor to minimize the chance that one of their commits causes them to violate Lose-work.

First, applications should try to crash as soon as possible after their bugs get triggered. Doing so shortens dangerous paths and thus lowers the probability of the application committing while executing on one. In order to move crashes sooner, processes can try to catch erroneous state by performing consistency checks. For example, a process could traverse its data structures looking for corruption, it could compute a checksum over some data, or it could inspect guard bands at the ends of its buffers and malloc'ed data. Voting amongst independent replicas is a general but expensive way to detect erroneous execution [27]. When a process fails one of these checks, it simply terminates execution, effectively crashing.

Although it is a good idea for processes to perform these consistency checks frequently, performing them right before committing is particularly important.

Applications may also be able to reduce the likelihood they will violate Lose-work by not committing all their state. Applications may have knowledge of which data absolutely must be preserved, and which data can be recomputed from an earlier (hopefully bug-free) state. Should a bug corrupt state that is not written to stable storage during commit, recomputing that state after a failure leaves open the possibility of not retriggering the bug.

Applications can also try to commit as infrequently as possible. When upholding Save-work, applications should do so with a protocol that commits less often and that leaves as much non-determinism as possible. Some applications may be able to add non-determinism to their execution, or they may be able to choose a non-deterministic algorithm over a deterministic one.

The application or the operating system may also be able to make some fixed non-deterministic events into transient

ones by increasing disk space or other application resource limits after a failure.

In Section 4 we will measure how often several applications violate Lose-work in the process of upholding Save-work.

3. Cost of Upholding Save-work

In Section 2.3, we presented the Save-work invariant, which applications can uphold to guarantee consistent recovery in the presence of stop failures. However, we have not talked about the performance penalty applications incur to uphold it. As mentioned above, executing commits can be expensive. It may be the case for real applications that adhering to Save-work may be prohibitively expensive. In this section we measure the performance penalty incurred for several real applications upholding Save-work.

For this experiment we have selected four real applications: *mvi*, *magic*, *xpilot* and *TreadMarks*. *mvi* is a public domain version of the well known Unix text editor *vi*. *magic* is a VLSI CAD tool. *xpilot* is a distributed, multi-user game. Finally, *TreadMarks* is a distributed shared memory system. Within *TreadMarks*'s shared memory environment we run an N-body simulation called Barnes-Hut.

Of these applications, all but *TreadMarks* are interactive. We chose mainly interactive applications for several reasons. First, interactive applications are important recipients of failure transparency (when these applications fail there is always an annoyed user nearby). Second, interactive applications have been little studied in recovery literature. Finally, interactive applications can be hard to recover: they have copious system state, non-determinism, and visible output, all of which requiring an able recovery system.

TreadMarks and *xpilot* are both distributed applications, while the others are single-process.

To recover these applications we run them on top of *Discount Checking*, a system designed to provide failure transparency efficiently using lightweight, full-process checkpoints [24]. *Discount Checking* is built on top of reliable memory provided by the Rio File Cache [9], and lightweight transactions provided by the Vista transaction library [23].

In order to preserve the full user-level state of a process, *Discount Checking* maps the process's entire address space into a segment of reliable memory managed by Vista. Vista traps updates to the process's address space using copy-on-write, and logs the before-images of updated regions to its persistent undo log. To capture the application state in the register file (which cannot be mapped into persistent memory), *Discount Checking* copies the register file into a persistent buffer at commit time. Thus, taking a checkpoint amounts to copying the register file, atomically discarding the undo log, and resetting page protections.

Although the steps outlined so far will allow *Discount Checking* to checkpoint and recover user-level state, Dis-

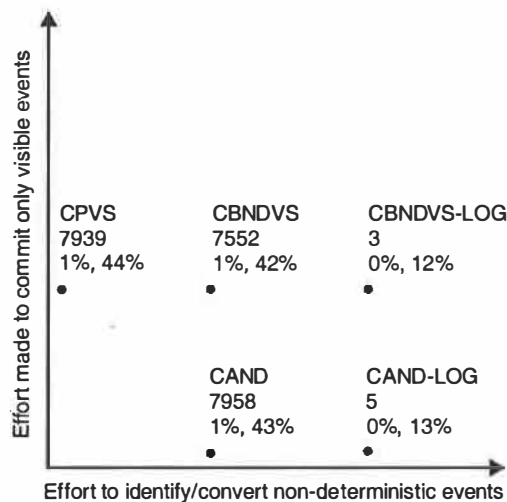
count Checking must also preserve and recover the application's kernel state. To capture system state, the library implements a form of copy-on-write for kernel data: it traps system calls, copies their parameter values into persistent buffers, and then uses those parameter values to directly reconstruct relevant kernel state during recovery. For more on the inner workings of *Discount Checking*, please see [24].

As mentioned in Section 2.4, there exist a large variety of protocols for upholding Save-work. In order to get a sense of which work best for our suite of applications, we implemented seven different protocols within *Discount Checking*. Our core protocols are CAND, CPVS, and CBNDVS, which we described in Section 2.4. Recall that CAND upholds Save-work by committing immediately after every non-deterministic event. CPVS commits just before all visible and send events. CBNDVS commits before a visible or send event if the process has executed a non-deterministic event since its last commit. We also added to *Discount Checking* the ability to log non-deterministic user input and message receive events to render them deterministic, as well as the ability to use two-phase commit so one process can safely pass a dependency on an uncommitted non-deterministic event to another process. Adding these techniques to our core protocols yielded an additional four protocols: CAND-LOG, CBNDVS-LOG, CPV-2PC, and CBNDV-2PC. For example, CAND-LOG executes a commit immediately after any non-deterministic event that has not been logged. CPV-2PC commits all processes whenever any process executes a visible, but does not need to commit before a process does a send.

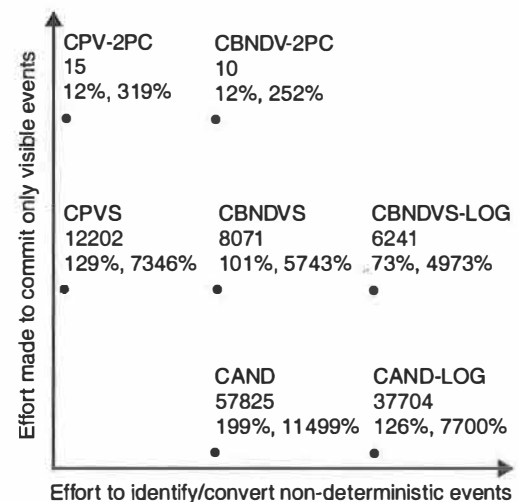
In order to implement these protocols, *Discount Checking* needs to get notification of an application's non-deterministic, visible, and send events. To learn of an application's non-deterministic events, *Discount Checking* intercepts a process's signals and non-deterministic system calls such as `gettimeofday`, `bind`, `select`, `read`, `recvmsg`, `recv`, and `recvfrom`. To learn of a process's visible and send events, *Discount Checking* intercepts calls to `write`, `send`, `sendto`, and `sendmsg`.

In addition to measuring the performance of our applications on *Discount Checking* on Rio, we wanted to get a sense of how our applications performed using a disk-based recovery system. We created a modified version of *Discount Checking* called DC-disk that wrote out a redo log synchronously to disk at checkpoint time. Although we did not add the code needed to let DC-disk truncate its redo log, or even properly recover applications, its overhead should be representative of what a lightweight disk-based recovery system can do.

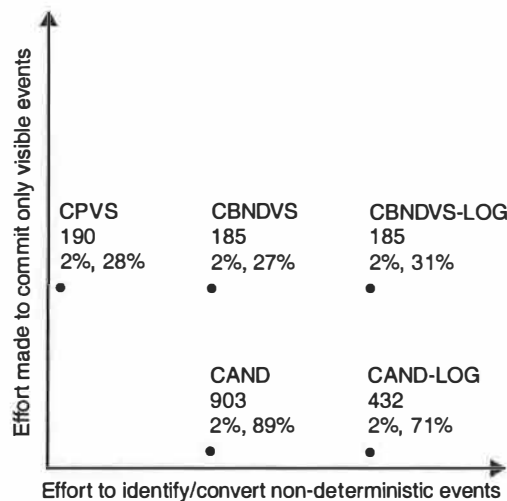
We ran our experiments on 400 MHz Pentium II computers each with 128 MB of memory (100 MHz SDRAM). Each machine runs FreeBSD 2.2.7 with Rio and is connected to a 100 Mb/s switched Ethernet. Rio was turned off when using DC-disk. Each computer has a single IBM Ultrastar



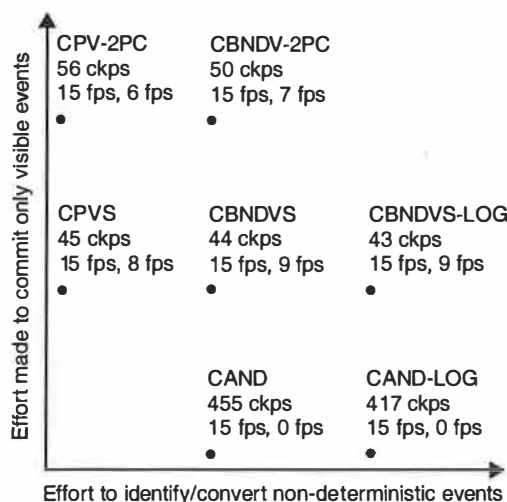
(a) *nvi*



(d) TreadMarks Barnes-Hut



(b) *magic*



(c) *xpilot*

Figure 8: Performance of several protocols for four applications. Each application has its own protocol space. At each point in each space, we list the protocol at that point, the number of checkpoints in the complete run of the application, and the runtime overhead for Discount Checking, and for DC-disk. For *xpilot* we list the protocol, number of checkpoints per second, followed by sustainable frame rate for Discount Checking and DC-disk. Full speed for *xpilot* is 15 frames per second.

DCAS-34330W ultra-wide SCSI disk. All points represent the average of five runs. The standard deviation for each data point was less than 1% of the mean for Discount Checking, and less than 4% of the mean for DC-disk. The distributed workloads (*TreadMarks* and *xpilot*) were both run on four computers. We simulate fast interactive rates by delaying 100 ms between each keystroke in *nvi* and by delaying 1 second between each mouse-generated command in *magic*.

We present the result of our runs in Figure 8. For each application we show the protocol space developed in Section 2.4. In each application's protocol space we plot the protocol used for each data point, and the number of checkpoints taken during the complete run of the application when running on that protocol. For each protocol's data point we also show the percent expansion in execution time that protocol caused compared to an unrecoverable version of the application, first for Discount Checking, then for DC-disk.

Because *xpilot* is a real-time, continuous program we report its performance as the frame rate it can sustain rather than runtime overhead. Higher frame rates indicate better interactivity, with full speed being 15 frames-per-second. *xpilot*'s number of checkpoints is given as the largest checkpointing frequency (in checkpoints per second) amongst its processes.

We can make a number of interesting observations based on these results. As expected, commit frequency generally decreases, and performance increases, with radial distance from the origin. The sole exception to this rule is *xpilot*, where having all processes commit whenever any one

of them wants to execute a visible event (as is done in protocols using two-phase commit) results in a net *increase* in commit frequency.

Despite the fact that several of these applications generate many commits, there is at least one protocol for each application with very low overhead for Discount Checking. We conclude that the cost of upholding Save-work using Discount Checking on these applications is low.

For all the interactive applications, the overhead of using DC-disk is not prohibitive. We see overhead of 12% and 27% for *nvi* and *magic* respectively. *xpilot* is able to sustain a usable 9 frames per second. On the other hand, no protocol for DC-disk was able to keep up with *TreadMarks*. From our experiments, we conclude that Save-work can be upheld with a disk-based recovery system for many interactive applications with reasonably low overhead.

We observe that the protocols that perform best for each application are the ones that exploit the infrequent class of events for that application in deciding when to commit. For example, *TreadMarks* has very few visible events, despite having copious non-deterministic and send events. For it, the 2PC protocols which let it commit only for the rare visible events are the big win.

While overhead is low for many applications, we can conceive of applications for which Save-work incurs a large performance overhead. These applications would have copious visible and non-deterministic events—that is, no rare class of events—and they would be compute bound rather than user bound. Applications that might fall into this category include interactive scientific or engineering simulation, online transaction processing, and medical visualization.

4. Measuring Conflict between the Save-work and Lose-work Invariants

Guaranteeing consistent recovery in the presence of stop and propagation failures requires upholding both the Save-work and Lose-work invariants. Unfortunately, some failure scenarios make it impossible to uphold both invariants simultaneously.

For example, consider the failure timeline shown in Figure 9. In this timeline, the application executes a transient non-deterministic event that causes it to execute down a code path containing a bug. The application eventually executes the buggy code (shown as “fault activation”), then correctly executes a visible event. After this visible event, the program crashes. Section 2.5’s coloring algorithm shows that the entire execution path from the transient non-deterministic event to the crash forms a dangerous path, along which the Lose-work invariant prohibits a commit. Unfortunately, the Save-work invariant specifically requires a commit between the transient non-deterministic event and the visible event. For this application, both invariants cannot be upheld simultaneously.

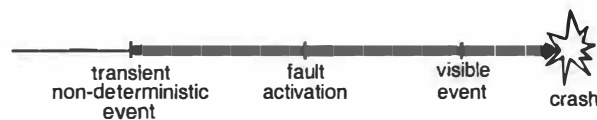


figure 9: Failure timeline in which the Save-work invariant and the lose-work invariant conflict. The shaded portion is the dangerous path.

Some applications may have bugs that prevent upholding Lose-work even without committing to uphold Save-work. For example, many applications contain repeatable bugs (so called, “Bohrbugs”[13]). With these faults it is possible to execute from the initial state of the program to the bug without ever executing a transient non-deterministic event. In other words, the dangerous path resulting from the bug extends all the way back to the initial state of the program. And since the initial state of any application is always committed, applications with Bohrbugs inherently violate Lose-work.

In this section, we endeavor to examine how often in practice faults cause a fundamental conflict between the Save-work and Lose-work invariants. Our focus is on software faults (both in the application and operating system), which field studies and everyday experience teach is the dominant cause of failures today [13].

4.1. Application faults

We would like to measure how often upholding Save-work forces an application to commit on a dangerous path, like the application depicted in Figure 9. We divide this problem into three subproblems. First, how often does an application bug create a dangerous path beginning at the start state of the application? As described above, this scenario arises from Bohrbugs in the application. Second, given an application fault that does depend on a transient non-deterministic event (a Heisenbug), how often is the application forced to commit between the transient non-deterministic event at the beginning of the dangerous path and the fault activation? Third, how often is the application forced to commit between the fault activation and the crash? We examine this third question first using a fault-injection study.

Our strategy is to force crashes of real applications, recover the applications, and measure after the fact whether any of their commits to uphold Save-work occurred between fault activation and the crash. We induce faults in the application by running a version of the application with changes in the source code to simulate a variety of programming errors. These errors include actions like overwriting random data in the stack or heap, changing the destination variable, neglecting to initialize a variable, deleting a branch, deleting a random line of source code, and off-by-one errors in conditions like \geq and $<$. See [6] for more information on our fault model. We only consider runs where the program crashes.

Fault Type	<i>nvi</i> Lose-work violations	<i>postgres</i> Lose-work violations
Stack bit flip	0%	35%
Heap bit flip	83%	92%
Destination reg	18%	0%
Initialization	4%	6%
Delete branch	81%	86%
Delete instruction	51%	13%
Off by one	24%	0%
Average	37%	33%

Table 1: Fraction of application faults in *nvi* and *postgres* that violate Lose-work by committing after the fault is activated. For each fault type we list the percent of crashes by that fault that commit after the fault is activated. Over all fault types, *nvi* and *postgres* commit after the fault activation for 37% and 33% of all crashes respectively.

Checkpointing and recovery for the applications is provided by Discount Checking using the CPVS protocol. CPVS is the best protocol possible for not violating Lose-work for non-distributed applications. For our experiments, we use two applications: the Unix text editor *nvi*, and *postgres*, a large, publicly available relational database. These two applications differ greatly in their code size and amount of data they touch while executing.

We detect a run in which the application commits between fault activation and the crash by instrumenting Discount Checking to log each fault activation and commit event. If the program commits after activating the fault, it has violated the Lose-work invariant. We also conduct an end-to-end check of this criteria by suppressing the fault activation during recovery, recovering the process, and trying to complete the run. As expected, we found that runs recovered from crashes if and only if they did not commit after fault activation.

We collected data from approximately 50 crashes for each fault type. Table 1 shows the fraction of crashes that violated the Lose-work invariant by committing after fault activation. For both *nvi* and *postgres*, approximately 35% of faults caused the process to commit along this portion of the dangerous path. While not included in the table, 7-9% of the runs did not crash but resulted in incorrect program output.

We next turn our attention to question one, namely, for what fraction of bugs does the dangerous path extend back to the initial state of the program? That is, of the bugs users encounter, what portion are deterministic (Bohrbugs), and what portion depend on a transient non-deterministic event (Heisenbugs)? Although it is difficult to measure this fraction directly, several prior studies have attempted to shed light on this issue.

Chandra and Chen showed that for *Apache*, *GNOME*, and *MySQL*, three large, publicly available software packages, only 5-15% of the bugs in the developer’s bug log were Heisenbugs (for shipping versions of the applications) [7]. The remaining bugs were Bohrbugs. Most of these deterministic bugs resulted from untested boundary conditions (e.g. an older version of *Apache* crashed when the URL was too long). Several other researchers have found a similarly low occurrence (5-29%) of application bugs that depend on transient non-deterministic events like timing [20, 29, 30]. Note that these results conflict with the conventional wisdom that mature code is populated mainly by Heisenbugs—it has been held that the easier-to-find Bohrbugs will be captured more often during development [13, 17]. It appears that for non-mission-critical applications, the current software culture tolerates a surprising number of deterministic bugs.

We have yet to tackle the second question, which asks how often an application is forced to commit on the dangerous path between the transient non-deterministic event and fault activation (see Figure 9). Unfortunately, we are unable to measure this frequency using our fault-injection technique because no realistic model exists for placing injected bugs relative to an application’s transient non-deterministic events. However, as we will see, the case for generic recovery from application failures is already sufficiently discouraging, even optimistically assuming no commits on this portion of the dangerous path.

We would like to compose these separate experimental results in order to illuminate the overarching question of this section. Our fault-injection study shows that *nvi* and *postgres* violate Lose-work in at least 35% of crashes from non-deterministic faults. If we assume the same distribution of deterministic and non-deterministic bugs in *nvi* and *postgres* as found in *Apache*, *GNOME*, and *MySQL* by Chandra and Chen, these non-deterministic faults make up only 5-15% of crashes. Therefore, Lose-work is upheld in at most 65% of 15%, or 10% of application crashes. Lose-work and Save-work appear to conflict in the remaining 90% of failures by these applications. While extrapolating other applications’ fault distributions to *nvi* and *postgres* is somewhat questionable, as is generalizing to all applications from the measurement of two, these preliminary results raise serious questions about the feasibility of generic recovery from propagation failures.

4.2. Operating systems faults

Although failures due to application faults appear to frequently violate Lose-work, we can hope for better news for faults in the operating system. In contrast to application faults, not all operating system faults cause propagation failures: some crash the system before they affect application state. Commits at any time by the application are okay in the presence of these stop failures. Thus if failures by the operat-

Fault Type	<i>nvi</i> failed recoveries	<i>postgres</i> failed recoveries
Stack bit flip	12%	10%
Heap bit flip	8%	6%
Destination reg	10%	0%
Initialization	16%	0%
Delete branch	26%	4%
Delete instruction	12%	4%
Off by one	22%	0%
Average	15%	3%

Table 2: Percent of OS faults in which *nvi* and *postgres* failed to recover. We list the percentage of crashes that led to failures during recovery for each fault type and over all failures.

ing system usually manifest as stop failures, we would rarely observe system failures causing Lose-work violations.

We wanted to measure the fraction of operating system failures for which applications are able to successfully recover. This fraction will include cases where the operating system experienced a stop failure, as well as cases in which the system failure was a propagation failure and the application did not violate Lose-work.

In order to perform this measurement, we again use a fault-injection study. This time we inject faults into the running kernel rather than into the application [9].

We again ran *nvi* and *postgres* with Discount Checking upholding Save-work using CPVS. For each run, we started the application and injected a particular type of fault into the kernel. We discarded runs in which neither the system nor the application crashed. If either the operating system or the application crashed, we rebooted the system and attempted to recover the application. We repeated this process until each fault type had induced approximately 50 crashes. The results of this experiment are shown in Table 2.

Of the 350 operating system crashes we induced for each application, we found that *nvi* failed to properly recover in 15% of crashes. *postgres* did better, only failing to recover 3% of the time. These numbers are encouraging: application-generic recovery is likely to work for operating systems failures, despite the challenge of upholding Lose-work.

If we assume that all propagation failures will violate Lose-work with the probabilities in Table 1 (regardless of whether the propagation failure began in the operating system or application), we can infer how often system failures manifest as propagation failures in our experiments. Combining our application crash results with our operating system crash results implies that for *nvi*, 41% of system failures were propagation failures. For *postgres*, 10% of system failures manifest as propagation failures. We hypothesize that the proportion of propagation failures differs for the two

applications because of the different rate at which they communicate with the operating system: the non-interactive version of *nvi* used in our crash tests executes almost 10 times as many system calls per second as *postgres* executes.

5. Related Work

Many fault-tolerant systems are constructed using transactions to aid recovery. Transactions simplify recovery by grouping separate operations into atomic units, reducing the number of states from which an application must recover after a crash. However, the programmer must still bear the responsibility for building recoverability into his or her applications, a task that is difficult even with transactions. We have focused on higher-level application-generic techniques that absolve programmers from adding recovery abilities to their software. However, we use transactions to implement our abstraction.

A number of researchers have endeavored to build systems that provide some flavor of failure transparency for stop failures [3, 4, 5, 11, 14, 21, 25, 26]. Our work extends their work by analyzing propagation failures as well.

The theory of distributed recovery has been studied at length [10]. Prior work has established that committed states in distributed systems must form a consistent cut to prevent orphan processes [8], that recoverable systems must preserve a consistent cut before visible events [28], and that non-determinism bounds the states preserved by commits [11, 16]. Our Save-work invariant is equivalent to the confluence of these prior results.

The Save-work invariant contributes to recovery theory by expressing the established rules for recovery in a single, elemental invariant. Viewing consistent recovery through the lens of Save-work, we exposed the protocol space and the relationships between the disparate protocols on it, as well as several new protocols.

To the best of our knowledge, no prior work has proposed an invariant for surviving propagation failures that relates all relevant events in a process, nor has any prior work attempted to evaluate the fraction of propagation failures for which consistent recovery is not possible.

CAND, CPVS, and CBNDVS all bear a resemblance to simple communication-induced checkpointing protocols (CIC) [1]. However there are some important differences. First, all CIC protocols assume no knowledge of application non-determinism. As a result, they are forced to roll back any process that has received a message from an aborted sender. Commits under these protocols serve primarily to limit rollback distance, and to prevent the domino effect. In contrast, our protocols all to varying degrees make use of knowledge of application non-determinism. Rather than abort the receivers of lost messages, they allow senders to deterministically regenerate the messages. Under our protocols, only failed processes are forced to roll back.

Recovery systems often depend on the assumption that applications will not commit faulty state—a so called “fail-stop assumption” [27]. Our examination of propagation failures amounts to a fine parsing of the traditional fail-stop assumption in which we consider a single commit’s ability to preserve not just past execution, but all future execution up to the next non-deterministic event. Making a fail-stop assumption in the presence of propagation failures is the same as assuming that applications can safely commit at any time without violating the Lose-work invariant.

6. Conclusion

The lure of operating systems that conceal failures is quite powerful. After all, what user or programmer wants to be burdened with the complexities of dealing with failures? Ideally, we could handle all those complexities once and for all in the operating system.

Our goal with this paper has been to explore the subject of failure transparency, looking at what it takes to provide it and exposing the circumstances where providing it is not possible. We find that providing failure transparency in general involves upholding two invariants, a Save-work invariant which constrains when an application must preserve its work before a failure, and a Lose-work invariant which constrains how much work the application has to throw away after a failure.

For stop failures, which do not require upholding Lose-work, the picture is quite rosy. We show that Save-work can be efficiently upheld for a variety of real applications. Using a transparent recovery system based on reliable memory, we find overheads of only 0-12% for our suite of real applications. We also find that disk-based recovery makes a credible go of it, with interactive applications experiencing only moderate overhead.

Unfortunately, the picture is somewhat bleaker for surviving propagation failures. Guaranteeing that an application can recover from a propagation failure requires upholding our Lose-work invariant, and Save-work and Lose-work can directly conflict for some fault scenarios. In our measurements of application faults in *nvi* and *postgres*, upholding Save-work causes them to violate Lose-work for at least 35% of crashes. Even worse, studies have suggested that 85-95% of application bugs today cause crashes that violate the Lose-work invariant by extending the dangerous path to the initial state.

We conclude that providing failure transparency for stop failures alone is feasible, but that recovering from propagation failures requires help from the application. Applications can help by performing better error detection, masking errors through N-version programming, reducing commit frequency by allowing the loss of some visible events, or reducing the comprehensiveness of the state saved by the recovery system. Our results point to interesting future work. Since pure application-generic recovery is not always possi-

ble, what is the proper balance between generic recovery services provided by the operating system and application-specific aids to recovery provided by the programmer?

7. Acknowledgements

Many people have thoughtfully contributed to this work. Thank you to the anonymous reviewers for their helpful feedback. Special thanks to Miguel Castro and Paul Resnick for their contributions to the theory. Thanks to George Dunlap for adding support for our user-level TCP to the XFree86 server. Finally, thanks to Christine Subietas, WRL’s administrator.

This research was supported in part by NSF grant MIP-9521386, AT&T Labs, IBM University Partnership Program, and Intel Corporation. Peter Chen was also supported by an NSF CAREER Award (MIP-9624869).

8. Software Availability

The Rio version of the FreeBSD 2.2.7 kernel, as well as Vista and Discount Checking are all available for download at <http://www.eecs.umich.edu/Rio>.

9. References

- [1] Lorenzo Alvisi, Elmootazbellah Elnozahy, Sriram Rao, Syed Amir Husain, and Asanka Del Mel. An Analysis of Communication-Induced Checkpointing. In *Proceedings of the 1999 International Symposium on Fault-Tolerant Computing (FTCS)*, June 1999.
- [2] Lorenzo Alvisi, Bruce Hoppe, and Keith Marzullo. Nonblocking and Orphan-Free Message Logging Protocols. In *Proceedings of the 1993 International Symposium on Fault-Tolerant Computing (FTCS)*, pages 145–154, 1994.
- [3] Joel F. Bartlett. A NonStop Kernel. In *Proceedings of the 1981 Symposium on Operating System Principles*, pages 22–29, December 1981.
- [4] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrman, and Wolfgang Oberle. Fault Tolerance Under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, February 1989.
- [5] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based Fault-tolerance. In *Proceedings of the 1995 Symposium on Operating Systems Principles*, pages 1–11, December 1995.
- [6] Subhachandra Chandra. *Evaluating the Recovery-Related Properties of Software Faults*. PhD thesis, University of Michigan, 2000.
- [7] Subhachandra Chandra and Peter M. Chen. Whither Generic Recovery from Application Faults? A Fault Study using Open-Source Software. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks (formerly FTCS)*, June 2000.

- [8] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States in Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.
- [9] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher M. Aycock, Gurushankar Rajamani, and David Lowell. The Rio File Cache: Surviving Operating System Crashes. In *Proceedings of the 1996 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1996.
- [10] Elmootazbellah N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Technical Report CMU-CS-99-148, Carnegie Mellon University, June 1999.
- [11] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit. *IEEE Transactions on Computers*, C-41(5):526–531, May 1992.
- [12] Jim Gray. *Operating Systems: An Advanced Course*. Springer-Verlag, 1978. Notes on Database Operating Systems.
- [13] Jim Gray. Why do computers stop and what can be done about it? In *Proceedings of the 1986 Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, January 1986.
- [14] Y. Huang, P. Y. Chung, C. M. R. Kintala, D. Liang, and C. Wang. NT-SwiFT: Software-implemented Fault Tolerance for Windows-NT. In *Proceedings of the 1998 USENIX WindowsNT Symposium*, August 1998.
- [15] David B. Johnson and Willy Zwaenepoel. Sender-Based Message Logging. In *Proceedings of the 1987 International Symposium on Fault-Tolerant Computing*, pages 14–19, July 1987.
- [16] David B. Johnson and Willy Zwaenepoel. Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing. In *Proceedings of the 1988 Symposium on Principles of Distributed Computing (PODC)*, pages 171–181, August 1988.
- [17] Zbigniew T. Kalbarczyk, Saurabh Bagchi, Keith Whisnant, and Ravishankar K. Iyer. Chameleon: A Software Infrastructure for Adaptive Fault Tolerance. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):560–579, June 1999.
- [18] Richard Koo and Sam Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.
- [19] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.
- [20] Inhwon Lee and Ravishankar K. Iyer. Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN Operating System. In *Proceedings of the 1993 International Symposium on Fault-Tolerant Computing (FTCS)*, pages 20–29, 1993.
- [21] David Lomet and Gerhard Weikum. Efficient Transparent Application Recovery in Client-Server Information Systems. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 460–471, June 1998.
- [22] David E. Lowell. *Theory and Practice of Failure Transparency*. PhD thesis, University of Michigan, 1999.
- [23] David E. Lowell and Peter M. Chen. Free Transactions with Rio Vista. In *Proceedings of the 1997 Symposium on Operating Systems Principles*, October 1997.
- [24] David E. Lowell and Peter M. Chen. Discount Checking: Transparent, Low-Overhead Recovery for General Applications. Technical Report CSE-TR-410-99, University of Michigan, December 1998.
- [25] James S. Plank, Micah Beck, and Gerry Kingsley. Libckpt: Transparent Checkpointing under Unix. In *Proceedings of the Winter 1995 USENIX Conference*, January 1995.
- [26] Michael L. Powell and David L. Presotto. PUBLISHING: A Reliable Broadcast Communication Mechanism. In *Proceedings of the 1983 Symposium on Operating Systems Principles*, October 1983.
- [27] Fred B. Schneider. Byzantine Generals in Action: Implementing Fail-Stop Processors. *ACM Transactions on Computer Systems*, 2(2):145–154, May 1984.
- [28] Robert E. Strom and Shaula Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.
- [29] Mark Sullivan and R. Chillarege. Software Defects and Their Impact on System Availability—A Study of Field Failures in Operating Systems. In *Proceedings of the 1991 International Symposium on Fault-Tolerant Computing*, June 1991.
- [30] Mark Sullivan and Ram Chillarege. A Comparison of Software Defects in Database Management Systems and Operating Systems. In *Proceedings of the 1992 International Symposium on Fault-Tolerant Computing*, pages 475–484, July 1992.
- [31] Yi-Min Wang. Maximum and minimum consistent global checkpoints and their applications. In *Proceedings of the 1995 Symposium on Reliable Distributed Systems*, pages 86–95, September 1995.

Design and Evaluation of a Continuous Consistency Model for Replicated Services *

Haifeng Yu Amin Vahdat

*Computer Science Department
Duke University
Durham, NC 27708*

{yhf, vahdat}@cs.duke.edu
<http://www.cs.duke.edu/~{yhf, vahdat}>

Abstract

The tradeoffs between consistency, performance, and availability are well understood. Traditionally, however, designers of replicated systems have been forced to choose from either strong consistency guarantees or none at all. This paper explores the semantic space between traditional strong and optimistic consistency models for replicated services. We argue that an important class of applications can tolerate relaxed consistency, but benefit from bounding the maximum rate of inconsistent access in an application-specific manner. Thus, we develop a set of metrics, *Numerical Error*, *Order Error*, and *Staleness*, to capture the consistency spectrum. We then present the design and implementation of TACT, a middleware layer that enforces arbitrary consistency bounds among replicas using these metrics. Finally, we show that three replicated applications demonstrate significant semantic and performance benefits from using our framework.

1 Introduction

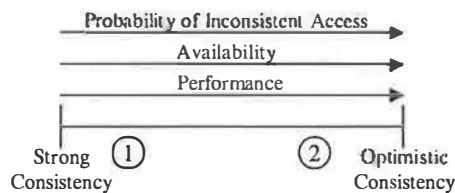
Replicating distributed services for increased availability and performance has been a topic of considerable interest for many years. Recently however, exponential increase in access to popular Web services provides us with concrete examples of the types of services that would benefit from replication, their requirements and semantics. One of the primary challenges to replicating network services is consistency across replicas. Providing strong consistency (e.g., one-copy serializability [4])

imposes performance overheads and limits system availability. Thus, a variety of optimistic consistency models [14, 15, 18, 31, 34] have been proposed for applications that can tolerate relaxed consistency. Such models require less communication, resulting in improved performance and availability.

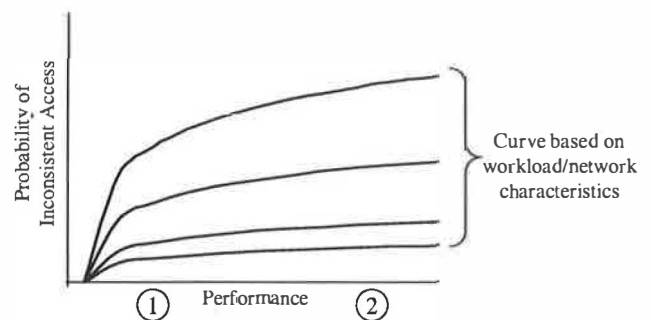
Unfortunately, optimistic models typically provide no bounds on the inconsistency of the data exported to client applications and end users. A fundamental observation behind this work is that there is a continuum between strong and optimistic consistency that is semantically meaningful for a broad range of network services. This continuum is parameterized by the maximum distance between a replica's local data image and some final image "consistent" across all replicas after all writes have been applied everywhere. For strong consistency, this maximum distance is zero, while for optimistic consistency it is infinite. We explore the semantic space in between these two extremes. For a given workload, providing a per-replica consistency bound allows the system to determine an expected probability, for example, that a write operation will conflict with a concurrent write submitted to a remote replica, or that a read operation observes the results of writes that must later be rolled back. No such analysis can be performed for optimistic consistency systems because the maximum level of inconsistency is unbounded.

The relationship between consistency, availability, and performance is depicted in Figure 1(a). In moving from strong consistency to optimistic consistency, application performance and availability increases. This benefit comes at the expense of an increasing probability that individual accesses will return inconsistent results, e.g., stale/dirty reads, or conflicting writes. In our work, we allow applications to bound the maximum probability/degree of inconsistent access in exchange for increased performance and availability. Figure 1(b) graphs different potential improvements in ap-

*This work is supported in part by the National Science Foundation (EIA-99772879, ITR-0082912). Vahdat is also supported by an NSF CAREER award (CCR-9984328). Additional information on the TACT project can be found at <http://www.cs.duke.edu/ari/issg/TACT/>.



(a)



(b)

Figure 1: a) The spectrum between strong and optimistic consistency as measured by a bound on the probability of inconsistent access. b) The tradeoff between consistency, availability, and performance depends upon application and network characteristics.

plication performance versus the probability of inconsistent access, depending on workload/network characteristics. Moving to the right in the figure corresponds to increased performance, while moving up in the figure corresponds to increased inconsistency. To achieve increased performance, applications must tolerate a corresponding increase in inconsistent accesses. The tradeoff between performance and consistency depends upon a number of factors, including application workload, such as read/write ratios, probability of simultaneous writes, etc., and network characteristics such as latency, bandwidth, and error rates. At the point labeled “1” in the consistency spectrum in Figure 1(b), a modest increase in performance corresponds to a relatively large increase in inconsistency for application classes corresponding to the top curve, perhaps making the tradeoff unattractive for these applications. Conversely, at point “2,” large performance increases are available in exchange for a relatively small increase in inconsistency for applications represented by the bottom curve.

Thus, the goals of this work are: i) to explore the issues associated with filling the semantic, performance, and availability gap between optimistic and strong consistency models, ii) to develop a set of metrics that allow a broad range of replicated services to conveniently and quantitatively express their consistency requirements, iii) to quantify the tradeoff between performance and consistency for a number of sample applications, and iv) to show the benefits of dynamically adapting consistency bounds in response to current network, replica, and client-request characteristics. To this end, we present the design, implementation, and evaluation of the TACT toolkit. TACT is a middleware layer that accepts specifications of application consistency requirements and mediates read/write access to an underlying data store. If an operation does not violate pre-specified consistency requirements, it proceeds locally (without contacting re-

mote replicas). Otherwise, the operation blocks until TACT is able to synchronize with one or more remote replicas (i.e., push or pull some subset of local/remote updates) as determined by system consistency requirements.

We propose three metrics, *Numerical Error*, *Order Error*, and *Staleness*, to bound consistency. Numerical error limits the total weight of writes that can be applied across all replicas before being propagated to a given replica. Order error limits the number of tentative writes (subject to reordering) that can be outstanding at any one replica, and staleness places a real-time bound on the delay of write propagation among replicas. Algorithms are then designed to bound each metric: Numerical error is bounded using a push approach based solely on local information; a write commitment algorithm combined with compulsory write pull enforces order error bound; and staleness is maintained using real-time vector. To evaluate the effectiveness of our system, we implement and deploy across the wide area three applications with a broad range of dynamically changing consistency requirements using the TACT toolkit: an airline reservation system, a distributed bulletin board service, and load distribution front ends to a Web server. Relative to strong consistency techniques, TACT improves the throughput of these applications by up to a factor of 10. Relative to weak consistency approaches, TACT provides strong semantic guarantees regarding the maximum inconsistency observed by individual read and write operations.

The rest of this paper is organized as follows. Section 2 describes the three network services implemented in the TACT framework to motivate our system architecture. Section 3 presents the system model and design we adopt for our target services. Next, Section 4 details the TACT architecture and Section 5 evaluates the performance of our three applications in the TACT framework. Finally, Section 6 places our work in the context

of related work and Section 7 presents our conclusions.

2 Applications

2.1 Airline Reservations

Our first application is a simple replicated airline reservation system that is designed to be representative of replicated E-commerce services that accept inquiries (searches) and purchase orders on a catalog. In our implementation, each server maintains a full replica of the flight information database and accepts user reservations and inquiries about seat availability. Consistency in this application is measured by the percentage of requests that access inconsistent results. For example, in the face of divergent replica images, a user may observe an available seat, when in fact the seat has been booked at another replica (false positive). Or a user may see a particular seat is booked when in fact, it is available (false negative). Intuitively, the probability of such events is proportional to the distance between the local replica image and some consistent final image.

One interesting aspect of this application is that its consistency requirements change dynamically based on client, network, and application characteristics. For instance, the system may wish to minimize the rate of inquiries/updates that observe inconsistent intermediate states for certain preferred clients. Requests from such clients may require a replica to update its consistency level (by synchronizing with other replicas) before processing the request or may be directed to a replica that maintains the requisite consistency by default. As another example, if network capacity (latency, bandwidth, error rate) among replicas is abundant, the absolute performance/availability savings may not be sufficient to outweigh the costs associated with weaker consistency models. Finally, the desired consistency level depends on individual application semantics. For airline reservations, the cost of a transaction that must be rolled back is fairly small when a flight is empty (one can likely find an alternate seat on the same flight), but grows as the flight fills.

2.2 Bulletin Board

The bulletin board application is a replicated message posting service modeled after more sophisticated services such as USENET. Messages are posted to individual replicas. Sets of updates are propagated among replicas, ensuring that all messages are eventually distributed to all replicas. This application is intended to be representative of interactive applications that often allow

concurrent read/write access under the assumption that conflicts are rare or can be resolved automatically.

Desirable consistency requirements for the bulletin board example include maintaining causal and/or total order among messages posted at different replicas. With causal order, a reply to a message will never appear before the original message at any replica. Total order ensures that all messages appear in the same order at all replicas, allowing the service to assign globally unique identifiers to each message. Another interesting consistency requirement for interactive applications, including the bulletin board, is to guarantee that at any time t , no more than k messages posted before t are missing from the local replica.

2.3 QoS Load Distribution

The final application implemented in our framework is a load distribution mechanism that provides Quality of Service (QoS) guarantees to a set of preferred clients. In this scenario, front-ends (as in LARD [27]) accept requests on behalf of two classes of clients, standard and preferred. The front ends forward requests to back end servers with the goal of reserving some pre-determined portion of server capacity for preferred clients. Thus, front ends allow a maximum number of outstanding requests (assuming homogeneous requests) at the back end servers. To determine the maximum number of "standard" requests that should be forwarded, each front end must communicate current access patterns to all other front ends.

One goal of designing such a system is to minimize the communication required to accurately distribute such load information among front ends. This QoS application is intended to be representative of services that independently track the same logical data value at multiple sites, such as a distributed sensor array, a load balancing system, or an aggregation query. Such services are often able to tolerate some bounded inaccuracy in the underlying values they track (e.g., average temperature, server load, or employee salary) in exchange for reduced communication overhead or power consumption.

3 System Design

In this section, we first describe the basic replication system model we assume, and then elaborate on the model and metrics we provide to allow applications to continuously specify consistency level.

3.1 System Model

For simplicity, we refer to application data as a data store, though the data can actually be stored in a

database, file system, persistent object, etc. The data store is replicated in full at multiple sites. Each replica accepts requests from users that can be made up of multiple primitive read/write operations. TACT mediates application read/write access to the data store. On a single replica, a read or write is isolated from other reads or writes during execution. Depending on the specified consistency requirements, a replica may need to contact other replicas before processing a particular request.

Replicas exchange updates by propagating writes. This can take the form of gossip messages [22], anti-entropy sessions [13, 28], group communication [5], broadcast, etc. We chose anti-entropy exchange as our write propagation method because of its flexibility in operating under a variety of network scenarios. Each write bears an accept stamp composed of a logical clock time [23] and the identifier of the accepting replica. Replicas deterministically order all writes based on this accept stamp. As in Bayou [28, 34], updates are procedures that check for conflicts with the underlying data store before being applied in a *tentative* state. A write is tentative until a replica is able to determine the write's final position in the serialization order, at which point it becomes *committed* through a write commitment algorithm (described below).

Each replica maintains a logical time vector, similar to that employed in Bayou and in Golding's work [13, 28, 34]. Briefly, each entry in the vector corresponds to the latest updates seen from a particular replica. The *coverage property* ensures that a replica has seen all updates (remote and local) up to the logical time corresponding to the minimum value in its logical time vector. This means the serialization positions of all writes with smaller logical time than that minimal value can be determined and thus those writes can be committed. Anti-entropy sessions update values in each replica's logical time vector based on the logical times/replicas of the writes exchanged. Note that writes may have to be reordered or rolled back before as dictated by serialization order.

While TACT's implementation of anti-entropy is not particularly novel, a primary aspect of our work is determining when and with whom to perform anti-entropy in order to guarantee a minimum level of consistency. Replicas may propagate writes to other replicas at any time through *voluntary anti-entropy*. However, we are more concerned with write propagation required for maintaining a desired level of consistency, called *compulsory anti-entropy*. Compulsory anti-entropy is necessary for the correctness of the system, while voluntary anti-entropy only affects performance.

3.2 A Continuous Consistency Model

In our consistency model, applications specify their application-specific consistency semantics using *conits*. A conit is a physical or logical unit of consistency, defined by the application. For example, in the airline reservation example, individual flights or blocks of seats on a flight may be defined as a conit. An interesting issue beyond the scope of this paper is setting the granularity of conits. The required per-conit accounting overhead (described below) argues for coarse conit granularity. Conversely, coarse-grained conits may introduce issues of false sharing as updates to one data item in a conit may reduce performance/availability for accesses to logically unrelated data items in the same conit.

For each conit, we quantify consistency continuously along a three-dimensional vector:

Consistency =

(Numerical Error, Order Error, Staleness)

Numerical Error bounds the discrepancy between the value of the conit relative to its value in the "final image." For applications that maintain numerical records, the semantics of this metric are straightforward. For other applications, however, application-specific weights (defaulting to one) can be assigned to individual writes. The weights are the relative importance of the writes, from the application's point of view. Numerical error then becomes the total weighted unseen writes for a conit. Based on application semantics, two different kinds of numerical error, absolute numerical error and relative numerical error, can be defined. *Order Error* measures the difference between the order that updates are applied to the local replica relative to their ordering in the eventual "final image." *Staleness* bounds the difference between the current time and the acceptance time of the oldest write on a conit not seen locally.

Figure 2 illustrates the definition of order error and numerical error in a simple example. Two replicas, *A* and *B*, accept updates on a conit containing two data items, *x* and *y*. The logical time vector for *A* is (24, 5). The coverage property implies that all writes in its log with logical time less than or equal to five are committed (indicated by the shaded box), leaving three tentative writes. Similarly, the logical time vector for *B* is (0, 17), meaning that both writes in its log are tentative. Order error bounds the maximum number of tentative writes at a replica, i.e., the maximum number of writes that may have to be reordered or rolled back because of activity at other replicas. In general, a lower bound on order error implies a lower probability that a read will observe an inconsistent intermediate state. In this example, if *A*'s order error is bounded to three, *A* must invoke the write commitment algorithm—performing compul-

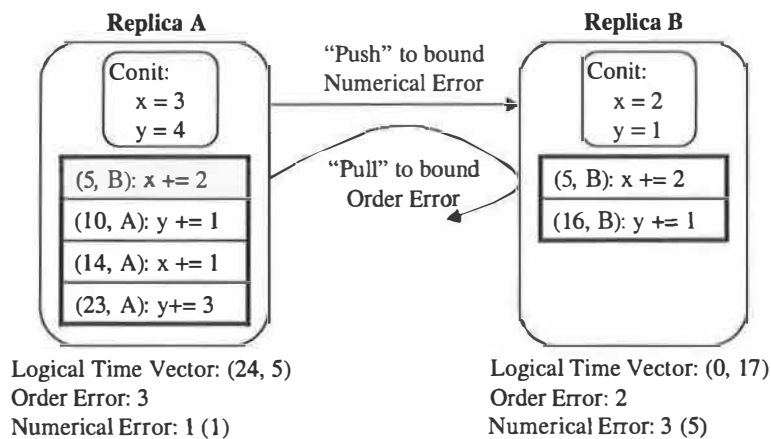


Figure 2: Example scenario for bounding order error and numerical error with two replicas.

sory anti-entropy to pull any necessary updates from *B* to reduce its number of tentative writes—before accepting any new writes.

Figure 2 also depicts the role of numerical error. Numerical error is the weight of all updates applied to a conit at *all* replicas not seen by the local replica. In the example, the weight of a write is set to be the update amount to either *x* or *y*, so that a “major” update is more important than a “minor” update. The replica *A* has not seen one update (with a weight of one) in this example, while *B* has not seen three updates (with a total weight of five). Note that order error can be relaxed or tightened using only local information. Bounding numerical error, on the other hand, relies upon the cooperation of all replicas. Thus, dynamically changing numerical error bounds requires the execution of a consensus algorithm.

One benefit of our model is that conit consistency can be bounded on a per-replica basis. Instead of enforcing a system-wide uniform consistency level, each replica can have its own independent consistency level for a conit. A simple analysis can show that as a replica relaxes its consistency while other replicas’ consistency levels remain unchanged, the total communication amount of that replica is reduced. For relaxed numerical error, it means other replicas can push writes to that replica less frequently, resulting in fewer incoming messages. Outgoing communication amount remains unchanged since that is determined by the consistency levels of other replicas. However, since numerical error is bounded using a push approach, if the replica is too busy to handle the outgoing communication, writes submitted to it will be delayed. Similar, if the replica relaxes order error and staleness, incoming communication amount will be decreased. Thus, one site may have poor network connectivity and limited processing power, making more relaxed consistency bounds appropriate for that replica.

Conversely, it may be cheap (from a performance and availability standpoint) to enforce stronger consistency at a replica with faster links and higher processing capacity. One interesting aspect of this model is that it potentially allows the system to route client requests to replicas with appropriate consistency bounds on a per-request basis. For instance, in the airline reservation system, requests from “preferred” clients may be directed to a replica that maintains higher consistency levels (reducing the probability of an inconsistent access).

When all three metrics are bounded to zero, our continuous consistency model reaches the strong consistency extreme of the spectrum, which is serializability [4] and external consistency [1, 12]. If no bounds are set for any of the metrics, there will be no consistency guarantees, similar to optimistic consistency systems. In moving from strong to optimistic consistency, applications bound the maximum logical “distance” between the local replica image and the (unknown) consistent image that contains all writes in serial order. This distance corresponds directly to the percentage chance that a read will observe inconsistent results or that a write will introduce a conflict. In the next section, we will demonstrate how our three applications employ these metrics to capture their consistency requirements. Based on our experience with TACT, we believe that the above metrics allow a broad range of applications to conveniently express their consistency requirements. Of course, the exact set of metrics is orthogonal to our goal of exporting a flexible, continuous, and dynamically tunable spectrum of consistency models to replicated services.

Application	Consistency Semantics	Conit Definition	Weight Definition	Metrics Capturing the Semantics
Bulletin Board	A. Message Ordering B. Unseen Messages C. Message Delay	A Newsgroup	(Subjective) Importance of a News Message	A. Order Error B. Absolute Numerical Error C. Staleness
Airline Reservation	A. Reservation Conflict Rate R B. Inconsistent Query Results	Seats on a Flight	Reservation: 1	A. Relative Numerical Error $R_{max} = 1 - 1/(1 + \gamma)$ $R_{avg} = (1 - 1/(1 + \gamma))/2$ B. Order Error and Staleness
QoS Load Distribution	A. Accuracy of Resource Consumption Info	Resource Consumption Info	Request Forward: 1 Request Return : -1	A. Relative Numerical Error

Table 1: Expressing high-level application-specific consistency semantics using the TACT continuous consistency model.

3.3 Expressing Application-specific Consistency Semantics through Conits

One important criteria for the evaluation of any consistency model is whether it captures the semantic requirements of a broad range of applications. Thus, in this section we describe the ways different consistency levels affect the semantics of the three representative applications described in Section 2 and explain how these semantics are captured by our model. Table 1 summarizes these application-specific consistency semantics and their expression using TACT, as detailed in the discussion below.

For the distributed bulletin board, one consistency requirement is the ordering of messages, which is captured by the order error metric. More specifically, for this application order error is the number of messages that may appear out of order at any replica. However, it is possible that such a bound is overly restrictive for unrelated messages, e.g., for two messages posted to different newsgroups. In this case, a conit can be defined for each newsgroup to more precisely specify ordering requirements. Another possible consistency requirement is the maximum number of remotely posted messages that are unseen by the local replica at a particular time. Our numerical error metric serves to express this type of semantics. Our model allows application-specific weights to be assigned to each write, allowing users to (subjectively) force the propagation of certain writes. A third consistency requirement for this application is message delay, that is, the delay between the time a message is posted and the time it is seen by all replicas. This requirement can be translated to staleness in a straight-forward manner.

Moving to the airline reservation example, one important manifestation of system consistency is the percentage of conflicting reservations. An interesting aspect of this application is TACT's ability to limit reservation conflict rate by bounding relative numerical error based on the application's estimate of available seats. To

simplify the discussion, we assume single seat reservations (though our model and implementation are more general) and define a conit over all seats on a flight with each reservation carrying a numerical weight of -1. Initially, the value of the conit is the total number of seats on the flight. As reservations come in, the value of the conit is the number of available seats in each replica's data store. Suppose reservations are randomly distributed among all available seats. For a reservation accepted by one replica, the probability that it conflicts with another remote (unseen) reservation is U/V , where U is the number of unseen reservations, and V is the number of available seats as seen by the local replica. Suppose V_{final} is the accurate count of available seats, such that $V_{final} = V - U$. Thus, the rate of conflicting reservations, R , equals $1 - V_{final}/V$. If γ bounds the maximum relative numerical error of the conit then, by definition, we have $-\gamma \leq 1 - V/V_{final} \Rightarrow V_{final} \geq 1/(1 + \gamma) \times V$. Thus, the upper bound on R , $R_{max} = 1 - V_{final}/V = 1 - 1/(1 + \gamma)$. Since in this example, V_{final} is always smaller than or equal to V , the average value of V_{final} should then be $(1 + 1/(1 + \gamma))/2 \times V$. Thus, the average rate of conflicting reservations, R_{avg} , equals to $1 - V_{final}/V = 1 - (1 + 1/(1 + \gamma))/2 = (1 - 1/(1 + \gamma))/2$. In Section 5, we present experimental results to verify this analysis. Non-random reservation behavior will result in a higher conflict rate than the above formula. However, applications can still reduce the expected/maximum conflict rate by specifying multiple conits per flight, e.g., multiple conits for first class versus coach seats or aisle versus window seats.

Other consistency semantics for the airline reservation example can be expressed using order error or staleness. For example, the system may wish to limit the percentage of queries that access an inconsistent image, i.e., see a multi-seat reservation that must later be rolled back because of a conflicting single-seat reservation at another replica. Such consistency semantics can be enforced by properly bounding the limit on order error (an analysis

is omitted for brevity).

In our third application, QoS load distribution, front ends estimate the total resource consumption for standard clients as the total number of outstanding standard requests on the back ends. This value also serves as the definition of a conit for this application. Front ends increase this value by 1 upon forwarding a request from a standard client and decrease it by 1 when the request returns. If this value exceeds a pre-determined resource consumption limit, front ends will not forward new standard client requests until resource consumption drops below this limit. The relative numerical error of each front end's estimate of resource consumption captures this application's consistency semantics — each front end is guaranteed that its estimate of resource consumption is accurate within a fixed bound. Note that this load balancing application is not concerned with order error (writes are interchangeable) or staleness (no need to synchronize if the mix of requests does not change).

3.4 Bounding Consistency Metrics

Given the assumed system model, we now describe in turn our algorithms for bounding numerical error, order error, and staleness. Note that the details and correctness proofs for our numerical error algorithms available separately [38].

The first algorithm, *Split-weight AE*, employs a “push” approach to bound absolute numerical error. It “allocates” the allowed positive and negative error for a server evenly to other servers. Each $server_i$ maintains two local variables x and y for $server_j, j \neq i$. Intuitively, the variable x is the total weight of negatively-weighted writes that $server_i$ accepts but has not been seen by $server_j$. $server_i$ has only conservative knowledge (called its *view*) of what writes $server_j$ has seen. The variable x is updated when $server_i$ accepts a new write with a negative weight or when $server_i$'s view is advanced. Similarly, the variable y records the total weight of positively-weighted writes. Suppose the absolute error bound on $server_j$ is α_j . In other words, we want to ensure that $|V_{final} - V_j| \leq \alpha_j$, where V_{final} is the consistent value and V_j is the value on $server_j$. To achieve this, $server_i$ makes sure that at all times, $x \geq -\alpha_j/(n-1)$ and $y \leq \alpha_j/(n-1)$, where n is the total number of servers in the system. This may require $server_i$ to push writes to $server_j$ before accepting a new write.

Split-Weight AE is pessimistic in the sense that $server_i$ may propagate writes to $server_j$ when not actually necessary. For example, the algorithm does not consider the case where negative weights and positive weights may offset each other. We developed another optimal algorithm, *Compound-Weight AE*, to ad-

dress this limitation at the cost of increased space overhead. However, simulations indicate that potential performance improvements do not justify the additional computational complexity and space overhead [38].

A third algorithm, *Inductive RE*, provides an efficient mechanism for bounding the relative error in numerical records. The algorithm transforms relative error into absolute error. Suppose the relative error bound for $server_j$ is γ_j , that is, we want to ensure $|1 - V_j/V_{final}| \leq \gamma_j$, equivalent to $|V_{final} - V_j| \leq \gamma_j \times V_{final}$. A naive transforming approach would use $\gamma_j \times V_{final}$ as the corresponding absolute error bound, requiring a consensus algorithm to be run to determine a new absolute error bound each time V_{final} changes.

Our approach avoids this cost by conservatively relying upon local information as follows. We observe that the current value V_i on any $server_i$ was properly bounded before the invocation of the algorithm and is an approximation of V_{final} . So $server_i$ may use V_i as an approximate norm to bound relative error for other servers. More specifically, for $server_i$, we know that $V_{final} - V_i \geq -\gamma_i \times V_{final}$, where γ_i is the relative error bound for $server_i$, which transforms to $V_{final} \geq V_i/(1 + \gamma_i)$. Using this information to substitute for V_{final} on the right-hand side in the inequality in the last paragraph produces:

$$|V_{final} - V_j| \leq \gamma_j \times \frac{V_i}{1 + \gamma_i}$$

Thus, to bound relative error, $server_i$ only needs to recursively apply Split-Weight AE, using $\gamma_j \times V_i/(1 + \gamma_i)$ as α_j . Note that while this approach greatly increases performance by eliminating the need to run a consensus algorithm among replicas, it uses local information ($V_i/(1 + \gamma_i)$) to approximate potentially unknown global information (V_{final}) in bounding relative error. Thus it behaves conservatively (bounding values more than strictly necessary) when relative error is high as will be shown in our evaluation of these algorithms in Section 5.

To bound order error on a per-conit basis, a replica first checks the number of tentative writes on a conit in its write log. If this number exceeds the order error limit, the replica invokes a write commitment algorithm to reduce the number of tentative writes in its write log. This algorithm operates as follows. The replica pulls writes from other replicas by performing compulsory anti-entropy sessions to advance its logical time vector, allowing it to commit some set of its tentative writes. In doing so, the replica ensures that it remains within a specified order error bound before accepting new tentative writes.

To bound the staleness of a replica, each server maintains a *real time vector*. This vector is similar to the log-

ical time vector, except that real time instead of logical time is used. A similar coverage property is preserved between the writes a server has seen and the real time vector. If A's real time vector entry corresponding to B is t , then A has seen all writes accepted by B before real time t . To bound staleness within l , a server checks whether $\text{current time} - t < l$ holds for each entry in the real time vector.¹ If the inequality does not hold for some entries, the server performs compulsory anti-entropy session with the corresponding servers, pulling writes from them, and advances the real time vector. This pull approach may appear to be less efficient than a push approach because of unnecessary polling when no updates are available. However, a push approach cannot bound staleness if there is no upper limit on network delay or processing time.

4 System Architecture

The current prototype of TACT is implemented in Java 1.2 using RMI for communication (e.g., for accepting read/write requests and for write propagation). TACT replicas are multi-threaded, thus if one write incurs compulsory write propagation, it will not block writes on other conits. We implemented a simple custom database for storing and retrieving data values, though our design and implementation is compatible with a variety of storage mechanisms.

Each TACT replica maintains a write log, and allows redo and undo on the write log. It is also responsible for all anti-entropy sessions with remote replicas. The system supports parallel anti-entropy sessions with multiple replicas, which can improve performance significantly for compulsory anti-entropy across the wide area. For increased efficiency, we also implement a one-round anti-entropy push. With standard anti-entropy, before a replica pushes writes to another replica, it first obtains the target replica's logical time vector to determine which writes to propagate. However, we found that this two-round protocol can add considerable overhead across the wide area, especially at stronger consistency levels (where the pushing replica has a fairly good notion of the writes seen by the target replica). Thus, we allow replicas to push writes using their local view as a hint, reducing two rounds of communication to one round at the cost of possibly propagating unnecessary writes. While the current implementation uses this one round protocol by default, dynamically switching between the variants based on the consistency level would be ideal.

TACT replicas also implement a consistency manager responsible for bounding numerical error, order error

and staleness. The variables needed by the Split-Weight AE and Inductive RE algorithms are maintained in hash tables to reduce space overhead and enable the system to potentially scale to thousands of conits.

In bounding numerical error, a replica may need to push a write to other replicas before the write can return, e.g., if a write has a weight that is larger than another replica's absolute error bound. There are two possible approaches for addressing this. One approach is a one-round protocol where the local site applies the write, propagates it to the necessary remote replicas, awaits acknowledgments, and finally returns. This one-round protocol is appropriate for applications where writes are interchangeable such as resource accounting/load balancing. For other applications, such as the airline reservation example, a reservation itself observes a consistency level (the probability it conflicts with another reservation submitted elsewhere). In such a case, a stronger two-round protocol is required where the replica first acquires remote data locks, pushes the write to remote replicas, and then returns after receiving all acknowledgments. Such a two-round protocol ensures the numerical error observed by a write is within bound at the time the update is submitted. In our prototype, both protocols are implemented and the application is allowed to choose based on its own requirements.

5 Experience and Evaluation

Given the description of our system architecture, we now discuss our experience in building the three applications described in Section 2 using the TACT infrastructure. We define conits and weights in these applications according to the analysis in Section 3.3. The experiments below focus on TACT's ability to bound numerical error and order error. While implemented in our prototype, we do not present experiments addressing staleness for brevity and because bounding staleness is well-studied, e.g., in the context of Web proxy caching [10].

5.1 Bulletin Board

For our evaluation of the bulletin board application, we deployed replicas at three sites across the wide area: Duke University (733 Mhz Pentium III/Solaris 2.8), University of Utah (350 Mhz Pentium II/FreeBSD 3.4) and University of California, Berkeley (167 Mhz Ultra I/Solaris 2.7). All data is collected on otherwise unloaded systems. Each submitted message is assigned a numerical weight of one (all messages are considered equally important).

We conduct a number of experiments to explore the behavior of the system at different points in the con-

¹We assume that server clocks are loosely synchronized.

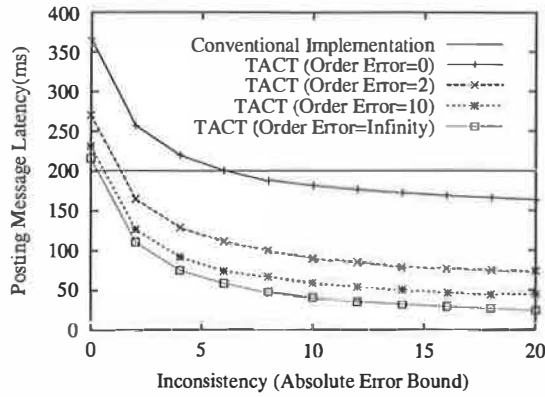


Figure 3: Average latency for posting messages to a replicated bulletin board as a function of consistency guarantees.

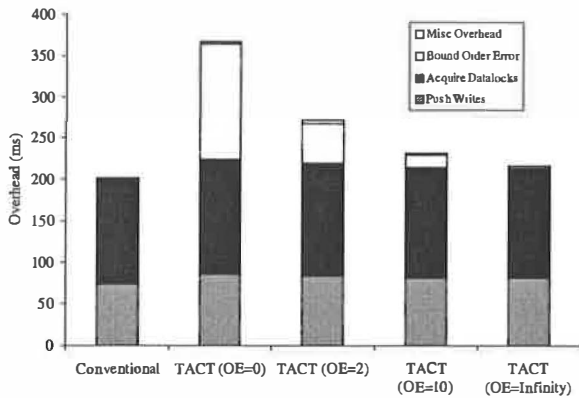


Figure 4: Breakdown of the overhead of posting a message under a number of scenarios.

sistency spectrum. Figure 3 plots the average latency for a client at Duke to post 200 messages as a function of the numerical error bound on the x-axis. For comparison, we also plot the average latency for a conventional implementation using a two-phase update protocol. For each write, this protocol first acquires necessary remote data locks, then propagates the update to all remote replicas. The figure shows how applications are able to continuously trade performance for consistency using TACT. As the numerical error bound increases, average latency decreases. Increasing allowable order error similarly produces a corresponding decrease in average latency. Relative to the conventional implementation, allowing each replica to have up to 20 unseen messages and leaving order error unbounded reduces average latency by a factor of 10.

One interesting aspect of Figure 3 is that TACT performs worse than the standard two-phase update protocol at the strong consistency end of the spectrum. To

investigate this overhead, Figure 4 summarizes the performance overheads associated with message posts using TACT at four points in the consistency spectrum (varying order error with numerical error set to zero) in comparison to the conventional two-phase update protocol. All five configurations incur approximately 130ms to sequentially (required to avoid deadlock) acquire data locks from two remote replicas and 80ms to push writes to these replicas in parallel. Since the cost of remote processing is negligible, this overhead comes largely from wide-area latency. Compared to the conventional implementation, TACT with zero numerical error and zero order error (i.e., same consistency level) incurs about 83% more overhead. This additional overhead stems from the additional 140ms to bound order error. This is an interesting side effect associated with our implementation. Our design decomposes consistency into two orthogonal components (numerical error and order error) that are bounded using two separate operations, doubling the number of wide-area round trip times. When order error and numerical error are both zero, TACT should combine the push and pull of write operations into a single step as a performance optimization, as is logically done by the conventional implementation. This idea is especially applicable if we use the recently proposed quorum approach[16, 17] to commit writes. A preliminary implementation of this optimization shows that TACT's overhead (at strong consistency) drops from 367ms to about 217ms, within 8% of the conventional approach.

5.2 Airline Reservation System

We now evaluate our implementation of the simple airline reservation system using TACT. Once again, we deployed three reservation replicas at Duke, Utah and Berkeley. We considered reservation requests for a single flight with 400 seats. Each client reservation request is for a randomly chosen seat on the flight. If a tentative reservation conflicts with a request at another replica, a merge procedure attempts to reserve a second seat on the same flight. If no seats are available, the reservation is discarded. A conit is defined over all seats on the flight, with an initial value of 400. Each reservation carries a numerical weight of -1.

In Section 3.3, we derived a relationship between the reservation conflict rate R and the relative error bound γ : $R_{max} = 1 - 1/(1 + \gamma)$ and $R_{avg} = (1 - 1/(1 + \gamma))/2$. We conduct the following experiment to verify that an application can limit the reservation conflict rate by simply bounding the relative numerical error. Figure 5 plots the measured conflicting reservation rate R , the computed upper bound R_{max} and the computed average rate R_{avg} as a function of relative numerical error. Order error and staleness are not bounded in these experiments.

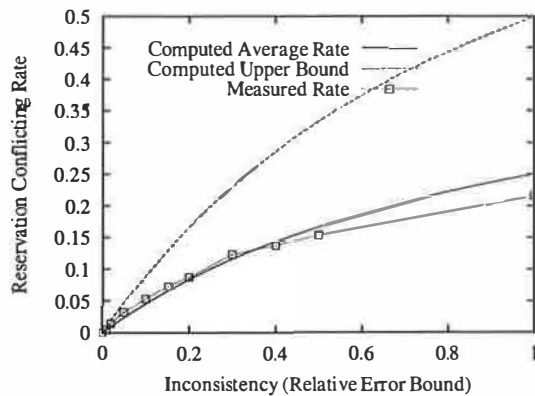


Figure 5: Percentage of conflicting reservations as a function of the bound on numerical error.

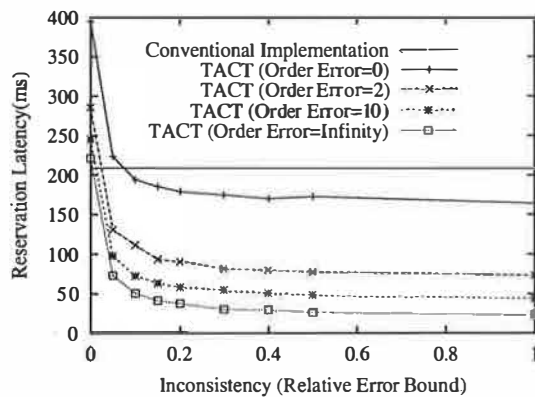


Figure 6: Average latency for making a reservation as a function of consistency guarantees.

The experiments are performed with two replicas on a LAN at Duke, each attempting to make 250 (random) reservations with the results averaged across four runs.

The measured conflict rate roughly matches the computed average rate and is always below the computed upper bound, demonstrating that numerical error can be used to bound conflicting accesses as shown by our analysis. Note that as the bound on relative error is relaxed, the discrepancy between the measured rate and the computed average rate gradually increases because of conservativeness inherent in the design of our Inductive RE algorithm (i.e., at relaxed consistency, our algorithm performs more write propagation than necessary). As described in Section 3, this conservative behavior greatly improves performance by allowing each replica to bound relative error using only local information.

The latency and throughput measurements, summarized in Figures 6 and 7 for airline reservations are similar to the bulletin board application described above.

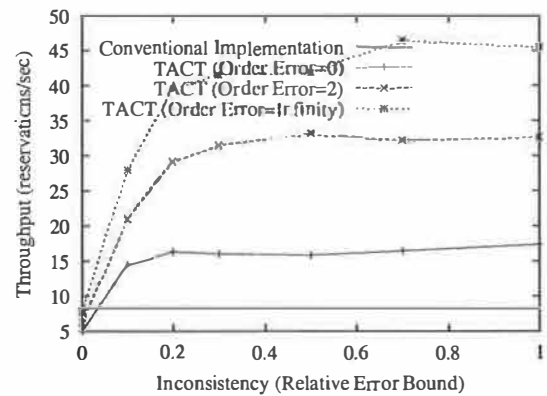


Figure 7: Update throughput for airline reservations as a function of consistency guarantees.

The latency experiments are run on the same wide-area configuration as the bulletin board. The plotted latency is the average observed by a single Duke client making 400 reservations. For throughput, we run two client threads at each of the replica sites, with each thread requesting $400/(2 \times 3) = 67$ (random) seats in a tight loop. We also plot the application's performance using a two-phase update protocol, showing the same trends as the results for the bulletin board application. As consistency is gradually relaxed, TACT achieves increasing performance by reducing the amount of required wide-area communication.

5.3 Quality of Service for Web Servers

For our final application, we demonstrate how TACT's numerical error bound can be used to accurately enforce quality of service (QoS) guarantees among Web servers distributed across the wide area. Recall that a number of front-end machines forward requests on behalf of both standard and preferred clients to back end servers. In our implementation, we use TACT to dynamically trade communication overhead in exchange for accuracy in measuring total resources consumed by standard clients. The front ends estimate the standard client resource consumption as the total number of outstanding standard requests on the back ends. If this resource consumption exceeds a pre-determined resource consumption limit, front ends will not forward new standard client requests until resource consumption drops below this limit. For simplicity, all our experiments are run on a local-area network at Duke on seven 733 Mhz Pentium III's running Solaris 2.8. Three front ends (each running on a separate machine) generate requests in a round robin fashion to three back end servers running Apache 1.3.12.

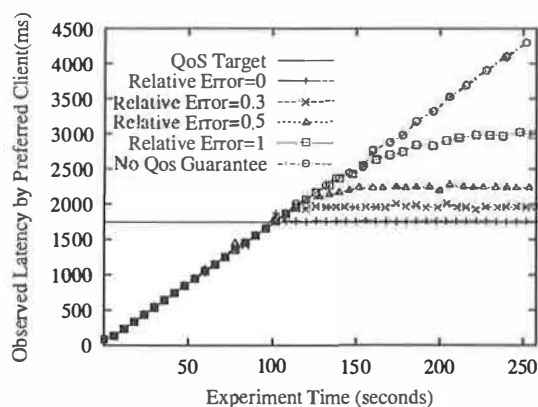


Figure 8: The average latency seen by a preferred client as a function of time.

Configuration	Consistency Messages
Relative Error=0	300
Relative Error=0.3	46
Relative Error=0.5	30
Relative Error=1	16
No QoS Guarantee	0

Table 2: The tradeoff between TACT-enforced numerical error and communication overhead.

For our experiments, the three front end machines generate an increasing number of requests from standard clients. As a whole, the system desires to bound the number of outstanding standard client requests to 150. A fourth machine, representing a preferred client, periodically polls a random back end to determine system latency. Each of the three front ends starts a new standard client every two seconds which then continuously requests the same dynamically generated Web page requiring 10ms of computation time. If all front ends had global knowledge of system state, each front end would start a total of 50 standard clients. However, depending on the bound placed on numerical error, front ends may in fact start more than this number (up to 130 in the experiment described below). For simplicity, no standard clients are torn down even if the system learns that too many (i.e., more than 150) are present in aggregate. Ideally, this aggregate number would oscillate around 150 with the amplitude of the oscillation being determined by the relative numerical bound.

Figure 8 depicts latency observed by the preferred client as a function of elapsed time (corresponding to the total number of standard clients making requests). At time 260, each front end has tried to spawn up to 130 standard clients. The curves show the average latency

observed by the preferred client for different bounds on numerical error. For comparison purposes, we also show the latency (1745ms) of a preferred client when there are exactly 150 outstanding standard client requests. In the first curve, labeled “Relative Error=0,” the system maintains strong consistency. Therefore, the front ends are able to enforce the resource limit strictly. The curve corresponding to a relative error of 0 flattens at 100 seconds (when three front ends have created a total of 150 standard clients) with latency very close to the ideal of 1745ms. As the bound on relative error is relaxed to 0.3, 0.5, and 1, the resource consumption limit for standard clients is more loosely enforced. The curve “No.QoS” plots the latency where no resource policy is enforced. Similar to the airline reservation application, the discrepancy between the relative error upper bound of 1 and the “No.QoS” curve stems from the conservativeness of the Inductive RE algorithm.

Table 2 quantifies the tradeoff between numerical error and communication overhead. Clearly, front ends can maintain near-perfect information about the load generated from other replicas at the cost of sending one message to all peers for each event that takes place. This is the case when zero numerical error is enforced by TACT: Each replica sends 50 messages to each of two remote replicas (for a total of 300) corresponding to the number of logical events that take place during the experiment. Once each front end starts 50 standard clients, strong consistency ensures that no further messages are necessary. Of course, such accuracy is typically not required by this application. Table 2 shows that communication overhead drops rapidly in exchange for some loss of accuracy. Note that this drop off will be more dramatic as the number of replicas is increased as a result of the all-to-all communication required to maintain strong consistency.

6 Related Work

The tradeoff between consistency and performance/availability is well understood [7, 8]. Many systems have been built at the two extremes of the consistency spectrum. Traditional replicated transactional databases use strong consistency (one-copy serializability [4]) as a correctness criterion. At the other end of the spectrum are optimistic systems such as Bayou [28, 34], Ficus [14], Rumor [15] and Coda [18]. In these systems, higher availability/performance is explicitly favored over strong consistency. Besides Bayou, none of the above systems provide support for different consistency levels. Bayou provides session guarantees [9, 33] to ensure that clients switching from one replica to another view a self-consistent version of

the underlying database. However, session guarantees do not provide any guarantees regarding the consistency level of a particular replica.

In [37], we present a position paper describing an earlier iteration of our consistency model, using different consistency metrics and concentrating on consistency/availability tradeoffs. A number of other efforts also attempt to numerically capture applications' consistency requirements. These techniques can be vaguely categorized into two classes: Relaxing consistency among replicas to reduce required communication (replica control) [3, 6, 19, 26, 32, 35] and relaxing consistency for transactions on a single site to allow increased concurrency on that site (concurrency control) [2, 20, 21, 29, 30, 36]. TACT is more closely related to replica control techniques. However, previous consistency models for replica control typically exploit the consistency semantics of a particular application class, abstracting its consistency requirements along a single dimension. Most of the proposed consistency metrics can be expressed within our model by constraining a subset of numerical error, order error, and staleness. Krishnakumar and Bernstein [19] propose the concept of an "N-ignorant" system, where a transaction runs in parallel with at most N conflicting transactions. By setting absolute numerical error bound to N and by assigning unit weights to writes, TACT demonstrates behavior similar to an "N-ignorant" system. Timed consistency [35] and delta consistency [32] address the lack of timing in traditional consistency models such as sequential consistency. These timed models can be readily expressed using our staleness metric. Quasi-copy caching [3] proposes four "coherency conditions," delay condition, frequency condition, arithmetic condition and version condition appropriate for read-only caching. TACT, on the other hand, is designed for more general read/write replication. Two recent efforts [6, 26] use metrics related to numerical error and staleness to measure database freshness. However, these systems do not provide mechanisms to bound data consistency using the proposed metrics. Relative to these efforts, our conit-based three-dimensional consistency model allows a wide range of services to dynamically express their consistency semantics based on application, network, and client-specific characteristics.

Concurrency control techniques using relaxed consistency models [2, 20, 21, 29, 30, 36] are related to replica control and TACT, in that consistency also needs to be quantified there. However, enforcing user-defined consistency levels is inherently easier in concurrency control than in replica control because in the former case most information needed to compute the amount of inconsistency is available on a single site. In other words, the consistency models do not need to consider "final

image," which might be unknown to all replicas. Since all our three metrics are related to "final image," none of them can be expressed using relaxed consistency models for concurrency control.

In fluid replication [24], clients are allowed to dynamically create service replicas to improve performance. Their study on when and where to create a service replica is complementary to our study on tunable consistency issues among replicas. Similar to Ladin's system [22], fluid replication supports three consistency levels: last-writer, optimistic and pessimistic. Our work focuses on capturing the spectrum between optimistic and pessimistic consistency models. Varying the frequency of reconciliation in fluid replication allows applications to adjust the "strength" of the last-writer and optimistic models. Bounding staleness in TACT has similar effects. However, as motivated earlier, staleness alone does not fully capture application-specific consistency requirements.

Fox and Brewer [11] argue that strong consistency and one-copy availability cannot be achieved simultaneously in the presence of network partitions. In the context of the Inktomi search engine, they show how to trade harvest for yield. Harvest measures the fraction of the data reflected in the response, while yield is the probability of completing a request. In TACT, we concentrate on consistency among service replicas, but a similar "harvest" concept can also be defined using our consistency metrics. For example, bounding numerical error has similar effects as guaranteeing a particular harvest. Finally, Olston and Widom [25] address tunable performance/precision tradeoffs in the context of aggregation queries over numerical database records.

7 Conclusions and Future Work

Traditionally, designers of replicated systems have been forced to choose between strong consistency, with its associated performance overheads, and optimistic consistency, with no guarantees regarding the probability of conflicting writes or stale reads. In this paper, we explore the space in between these two extremes. We present a continuous consistency model where application designers can bound the maximum distance between the local data image and some final consistent state. This space is parameterized by three metrics, *Numerical Error*, *Order Error*, and *Staleness*. We show how TACT, a middleware layer that enforces consistency bounds among replicas, allows applications to dynamically trade consistency for performance based on current service, network, and request characteristics. A performance evaluation of three replicated applications, an airline reservation system, a bulletin board, and a QoS Web

service, implemented using TACT demonstrates significant semantic and performance benefits relative to traditional approaches.

We are investigating a number of interesting questions posed by the TACT consistency model. We are currently working on both theoretical and practical issues associated with trading system consistency for availability. Theoretically, is there an upper bound on availability given a consistency level with particular numerical error, order error and staleness? Practically, how close to this upper bound can the TACT prototype provide dynamically tunable consistency and availability? Similarly, can TACT adaptively set application consistency levels in response to changing wide-area network performance characteristics using application-specified targets for minimum performance, availability, and consistency?

Acknowledgments

We would like to thank David Becker for maintaining the experimental infrastructure at Duke. Jay Lepreau and Stephen Clawson were very helpful in providing access to the Utah computing resources available at the University of Utah Network Testbed (supported by NSF grant ANI-00-82493, DARPA/AFRL grant F30602-99-1-0503, and Cisco). Jeff Chase and Doug Terry provided a number of insightful comments on the TACT design. Finally, the comments of David Culler, Robert Grimm, Brian Noble, and the anonymous referees greatly improved the presentation of this paper.

References

- [1] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, May 1995.
- [2] D. Agrawal, A. E. Abbadi, and A. K. Singh. Consistency and Orderability: Semantics-Based Correctness Criteria for Databases. *ACM Transactions on Database Systems*, September 1993.
- [3] Rafael Alonso, Daniel Barbara, and Hector Garcia-Molina. Data Caching Issues in an Information Retrieval System. *ACM Transactions on Database Systems*, September 1990.
- [4] Phil Bernstein and Nathan Goodman. The Failure and Recovery Problem for Replicated Distributed Databases. *ACM Transactions on Database Systems*, December 1984.
- [5] Ken P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):36–53, 1993.
- [6] Junghoo Cho and Hector Garcia-Molina. Synchronizing a Database to Improve Freshness. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, May 2000.
- [7] Brian Coan, Brian Oki, and Elliot Kolodner. Limitations on Database Availability When Networks Partition. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing*, pages 187–194, August 1986.
- [8] Susan Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in Partitioned Networks. *Computing Survey*, 17(3), 1985.
- [9] W. Keith Edwards, Elizabeth Mynatt, Karin Petersen, Mike Spreitzer, Douglas Terry, and Marvin Theimer. Designing and implementing asynchronous collaborative applications with bayou. In *Proceedings of 10th ACM Symposium on User Interface Software and Technology*, October 1997.
- [10] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2068, January 1997.
- [11] Armando Fox and Eric Brewer. Harvest, Yield, and Scalable Tolerant Systems. In *Proceedings of HOTOS-VII*, March 1999.
- [12] D. K. Gifford. Information Storage in a Decentralized Computer System. Technical Report CSL-81-8, Xerox PARC, 1983.
- [13] Richard Golding. *Weak-Consistency Group Communication and Membership*. PhD thesis, University of California, Santa Cruz, December 1992.
- [14] R. Guy, J. Heidemann, W. Mak, T. Page Jr., G. Popek, and D. Rothmeier. Implementation of the Ficus Replicated File System. In *Proceedings Summer USENIX Conference*, June 1990.
- [15] R. G. Guy, P. Reiher, D. Ratner, M. Gunter, W. Ma, and G. J. Popek. Rumor: Mobile Data Access Through Optimistic Peer-to-Peer Replication. In *Proceedings of the 17th International Conference on Conceptual Modeling (ER'98)*, November 1998.
- [16] J. Holliday, R. Steinke, D. Agrawal, and A. El Abbadi. Epidemic Quorums for Managing Replicated Data. In *Proceedings of the 19th IEEE International Performance, Computing, and Communications Conference*, February 2000.
- [17] Peter Keleher. Decentralized Replicated-Object Protocols. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, April 1999.
- [18] James J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.
- [19] Narayanan Krishnakumar and Arthur Bernstein. Bounded Ignorance: A Technique for Increasing Concurrency in a Replicated System. *ACM Transactions on Database Systems*, 19(4), December 1994.

- [20] Tei-Wei Kuo and Aloysius K. Mok. Application Semantics and Concurrency Control of Real-Time Data-Intensive Applications. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1992.
- [21] Tei-Wei Kuo and Aloysius K. Mok. SSP: A Semantics-Based Protocol for Real-Time Data Access. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1993.
- [22] R. Ladin, B. Liskov, L. Shirira, and S. Ghemawat. Providing Availability Using Lazy Replication. *ACM Transactions on Computer Systems*, 10(4):360–391, 1992.
- [23] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.
- [24] Brian Noble, Ben Fleis, and Minkyong Kim. A Case for Fluid Replication. In *Proceedings of the 1999 Network Storage Symposium (Netstore)*, October 1999.
- [25] Chris Olston and Jennifer Widom. Bounded Aggregation: Offering a Precision-Performance Tradeoff in Replicated Systems. Technical report, Computer Science Department, Stanford University, 1999. <http://www-db.stanford.edu/pub/papers/trapp-ag.ps>.
- [26] Esther Pacitti, Eric Simon, and Rubens Melo. Improving Data Freshness in Lazy Master Schemes. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems*, May 1998.
- [27] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.
- [28] Karin Petersen, Mike Spreitzer, Douglas Terry, Marvin Theimer, and Alan Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP-16)*, pages 288–301, October 1997.
- [29] Calton Pu, Wenwey Hseush, Gail E. Kaiser, Kun-Lung Wu, and Philip S. Yu. Distributed Divergence Control for Epsilon Serializability. In *Proceedings of the International Conference on Distributed Computing Systems*, 1993.
- [30] Calton Pu and Avraham Leff. Replication Control in Distributed System: An Asynchronous Approach. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, May 1991.
- [31] Yasushi Saito, Brian Bershad, and Hank Levy. Manageability, Availability and Performance in Porcupine: A Highly Scalable Internet Mail Service. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, December 1999.
- [32] Aman Singla, Umakishore Ramachandran, and Jessica Hodgins. Temporal Notions of Synchronization and Consistency in Beehive. In *Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures*, June 1997.
- [33] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welch. Session Guarantees for Weekly Consistent Replicated Data. In *Proceedings 3rd International Conference on Parallel and Distributed Information System*, September 1994.
- [34] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 172–183, December 1995.
- [35] Francisco Torres-Rojas, Mustaque Ahamad, and Michel Raynal. Timed Consistency for Shared Distributed Objects. In *Proceedings of the 18th ACM Symposium on Principle of Distributed Computing*, May 1999.
- [36] M. H. Wong and D. Agrawal. Tolerating Bounded Inconsistency for Increasing Concurrency in Database Systems. In *Proceedings of the 11th Symposium on Principles of Database Systems*, June 1992.
- [37] Haifeng Yu and Amin Vahdat. Building Replicated Internet Services Using TACT: A Toolkit for Tunable Availability and Consistency Tradeoffs. In *Proceedings of the Second International Workshop on Advanced Issues of E-Commerce and Web-based Information Systems*, June 2000.
- [38] Haifeng Yu and Amin Vahdat. Efficient Numerical Error Bounding for Replicated Network Services. In *Proceedings of the 26th International Conference on Very Large Databases (VLDB)*, September 2000.

Scalable, Distributed Data Structures for Internet Service Construction

Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler

The University of California at Berkeley

{gribble,brewer,jmh,culler}@cs.berkeley.edu

Abstract

This paper presents a new persistent data management layer designed to simplify cluster-based Internet service construction. This self-managing layer, called a distributed data structure (DDS), presents a conventional single-site data structure interface to service authors, but partitions and replicates the data across a cluster. We have designed and implemented a distributed hash table DDS that has properties necessary for Internet services (incremental scaling of throughput and data capacity, fault tolerance and high availability, high concurrency, consistency, and durability). The hash table uses two-phase commits to present a coherent view of its data across all cluster nodes, allowing any node to service any task. We show that the distributed hash table simplifies Internet service construction by decoupling service-specific logic from the complexities of persistent, consistent state management, and by allowing services to inherit the necessary service properties from the DDS rather than having to implement the properties themselves. We have scaled the hash table to a 128 node cluster, 1 terabyte of storage, and an in-core read throughput of 61,432 operations/s and write throughput of 13,582 operations/s.

1 Introduction

Internet services are successfully bringing infrastructural computing to the masses. Millions of people depend on Internet services for applications like searching, instant messaging, directories, and maps, and also to safeguard and provide access to their personal data (such as email and calendar entries). As a direct consequence of this increasing user dependence, today's Internet services must possess many of the same properties as the telephony and power infrastructures. These *service properties* include the ability to scale to large, rapidly growing user populations, high availability in the face of partial failures, strictly maintaining the consistency of users' data, and operational manageability.

It is challenging for a service to achieve all of these properties, especially when it must manage large amounts of persistent state, as this state must

remain available and consistent even if individual disks, processes, or processors crash. Unfortunately, the consequences of failing to achieve the properties are harsh, including lost data, angry users, and perhaps financial liability. Even worse, there appear to be few reusable Internet service construction platforms (or data management platforms) that successfully provide all of the properties.

Many projects and products propose using software platforms on clusters to address these challenges and to simplify Internet service construction [1, 2, 6, 15]. These platforms typically rely on commercial databases or distributed file systems for persistent data management, or they do not address data management at all, forcing service authors to implement their own service-specific data management layer. We argue that databases and file systems have not been designed with Internet service workloads, the service properties, and cluster environments specifically in mind, and as a result, they fail to provide the right scaling, consistency, or availability guarantees that services require.

In this paper, we bring scalable, available, and consistent data management capabilities to cluster platforms by designing and implementing a reusable, cluster-based storage layer, called a *distributed data structure (DDS)*, specifically designed for the needs of Internet services. A DDS presents a conventional single site in-memory data structure interface to applications, and durably manages the data behind this interface by distributing and replicating it across the cluster. Services inherit the aforementioned service properties by using a DDS to store and manage all persistent service state, shielding service authors from the complexities of scalable, available, persistent data storage, thus simplifying the process of implementing new Internet services.

We believe that given a small set of DDS types (such as a hash table, a tree, and an administrative log), authors will be able to build a large class of interesting and sophisticated servers. This paper describes the design, architecture, and implementation of one such distributed data structure (a distributed hash table built in Java). We evaluate

its performance, scalability and availability, and its ability to simplify service construction.

1.1 Clusters of Workstations

In [15], it is argued that clusters of workstations (commodity PC's with a high-performance network) are a natural platform for Internet services. Each cluster node is an independent failure boundary, which means that replicating computation and data can provide fault tolerance. A cluster permits incremental scalability: if a service runs out of capacity, a good software architecture allows nodes to be added to the cluster, linearly increasing the service's capacity. A cluster has natural parallelism: if appropriately balanced, all CPUs, disks, and network links can be used simultaneously, increasing the throughput of the service as the cluster grows. Clusters have high throughput, low latency redundant system area networks (SAN) that can achieve 1 Gb/s throughput with 10 to 100 μ s latency.

1.2 Internet Service Workloads

Popular Internet services process hundreds of millions of tasks per day. A task is usually "small", causing a small amount of data to be transferred and computation to be performed. For example, according to press releases, Yahoo (<http://www.yahoo.com>) serves 625 million page views per day. Randomly sampled pages from the Yahoo directory average 7KB of HTML data and 10KB of image data. Similarly, AOL's web proxy cache (<http://www.aol.com>) handles 5.2 billion web requests per day, with an average response size of 5.5 KB. Services often take hundreds of milliseconds to process a given task, and their responses can take many seconds to flow back to clients over what are predominantly low bandwidth last-hop network links [19]. Given this high task throughput and non-negligible latency, a service may handle thousands of tasks simultaneously. Human users are typically the ultimate source of tasks; because users usually generate a small number of concurrent tasks (e.g., 4 parallel HTTP GET requests are typically spawned when a user requests a web page), the large set of tasks being handled by a service are largely independent.

2 Distributed Data Structures

A distributed data structure (DDS) is a self-managing storage layer designed to run on a cluster of workstations [2] and to handle Internet service workloads. A DDS has all of the previously mentioned service properties: high throughput, high concurrency, availability, incrementally scalability, and strict consistency of its data. Service authors see the interface to a DDS as a conventional data

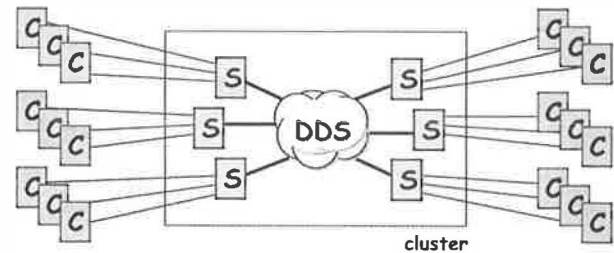


Figure 1: **High-level view of a DDS:** a DDS is a self-managing, cluster-based data repository. All service instances (S) in the cluster see the same consistent image of the DDS; as a result, any WAN client (C) can communicate with any service instance.

structure, such as a hash table, a tree, or a log. Behind this interface, the DDS platform hides all of the mechanisms used to access, partition, replicate, scale, and recover data. Because these complex mechanisms are hidden behind the simple DDS interface, authors only need to worry about service-specific logic when implementing a new service. The difficult issues of managing persistent state are handled by the DDS platform.

Figure 1 shows a high-level illustration of a DDS. All cluster nodes have access to the DDS and see the same consistent image of the DDS. As long as services keep all persistent state in the DDS, any service instance in the cluster can handle requests from any client, although we expect clients will have affinity to particular service instances to allow session state to accumulate.

The idea of having a storage layer to manage durable state is not new, of course; databases and file systems have done this for many decades. The novel aspects of a DDS are the level of abstraction that it presents to service authors, the consistency model it supports, the access behavior (concurrency and throughput demands) that it presupposes, and its many design and implementation choices that are made based on its expected runtime environment and the types of failures that it should withstand. A direct comparison between databases, distributed file systems, and DDS's helps to show this.

Relational database management systems (RDBMS): an RDBMS offers extremely strong durability and consistency guarantees, namely ACID properties derived from the use of transactions [18], but these ACID properties can come at high cost in terms of complexity and overhead. As a result, Internet services that rely on RDBMS backends typically go to great lengths to reduce the workload presented to the RDBMS, using techniques such as query caching in front ends [15, 21, 32]. RDBMS's offer a high degree of data independence, which is a powerful abstraction that adds addi-

tional complexity and performance overhead. The many layers of most RDBMS's (such as SQL parsing, query optimization, access path selection, etc.) permit users to decouple the logical structure of their data from its physical layout. This decoupling allows users to dynamically construct and issue queries over the data that are limited only by what can be expressed in the SQL language, but data independence can make parallelization (and therefore scaling) hard in the general case. From the perspective of the service properties, an RDBMS always chooses consistency over availability: if there are media or processor failures, an RDBMS can become unavailable until the failure is resolved, which is unacceptable for Internet services.

Distributed file systems: file systems have less strictly defined consistency models. Some (e.g., NFS [31]) have weak consistency guarantees, while others (e.g., Frangipani [33] or AFS [12]) guarantee a coherent filesystem image across all clients, with locking typically done at the granularity of files. The scalability of distributed file systems similarly varies; some use centralized file servers, and thus do not scale. Others such as xFS [3] are completely serverless, and in theory can scale to arbitrarily large capacities. File systems expose a relatively low level interface with little data independence; a file system is organized as a hierarchical directory of files, and files are variable-length arrays of bytes. These elements (directories and files) are directly exposed to file system clients; clients are responsible for logically structuring their application data in terms of directories, files, and bytes inside those files.

Distributed data structures (DDS): a DDS has a strictly defined consistency model: all operations on its elements are atomic, in that any operation completes entirely, or not at all. DDS's have one-copy equivalence, so although data elements in a DDS are replicated, clients see a single, logical data item. Two-phase commits are used to keep replicas coherent, and thus all clients see the same image of a DDS through its interface. Transactions across multiple elements or operations are not currently supported: as we will show later, many of our current protocol design decisions and implementation choices exploit the lack of transactional support for greater efficiency and simplicity. There are Internet services that require transactions (e.g. for e-commerce); we can imagine building a transactional DDS, but it is beyond the scope of this paper, and we believe that the atomic single-element updates and coherence provided by our current DDS are strong enough to support interesting services.

A DDS's interface is more structured and at a higher level than that of a file system. The granularity of an operation is a complete data structure

element rather than an arbitrary byte range. The set of operations over the data in a DDS is fixed by a small set of methods exposed by the DDS API, unlike an RDBMS in which operations are defined by the set of expressible declarations in SQL. The query parsing and optimization stages of an RDBMS are completely obviated in a DDS, but the DDS interface is less flexible and offers less data independence.

In summary, by choosing a level of abstraction somewhere in between that of an RDBMS and a file system, and by choosing a well-defined and simple consistency model, we have been able to design and implement a DDS with all of the service properties. It has been our experience that the DDS interfaces, although not as general as SQL, are rich enough to successfully build sophisticated services.

3 Assumptions and Design Principles

In this section of the paper, we present the design principles that guided us while building our distributed hash table DDS. We also state a number of key assumptions we made regarding our cluster environment, failure modes that the DDS can handle, and the workloads it will receive.

Separation of concerns: the clean separation of service code from storage management simplifies system architecture by decoupling the complexities of state management from those of service construction. Because persistent service state is kept in the DDS, service instances can crash (or be gracefully shut down) and restart without a complex recovery process. This greatly simplifies service construction, as authors need only worry about service-specific logic, and not the complexities of data partitioning, replication, and recovery.

Appeal to properties of clusters: in addition to the properties listed in section 1.1, we require that our cluster is physically secure and well-administered. Given all of these properties, a cluster represents a carefully controlled environment in which we have the greatest chance of being able to provide all of the service properties. For example, its low latency SAN (10-100 μ s instead of 10-100 ms for the wide-area Internet) means that two-phase commits are not prohibitively expensive. The SAN's high redundancy means that the probability of a network partition can be made arbitrarily small, and thus we need not consider partitions in our protocols. An uninterruptible power supply (UPS) and good system administration help to ensure that the probability of system-wide simultaneous hardware failure is extremely low; we can thus rely on data being available in more than one failure boundary (i.e., the physical memory or disk of more than one

node) while designing our recovery protocols.¹

Design for high throughput and high concurrency: given the workloads presented in section 1.2, the control structure used to effect concurrency is critical. Techniques often used by web servers, such as process-per-task or thread-per-task, do not scale to our needed degree of concurrency. Instead, we use asynchronous, event-driven style of control flow in our DDS, similar to that espoused by modern high performance servers [5, 20] such as the Harvest web cache [8] and Flash web server [28]. A convenient side-effect of this style is that layering is inexpensive and flexible, as layers can be constructed by chaining together event handlers. Such chaining also facilitates interposition: a “middleman” event handler can be easily and dynamically patched between two existing handlers. In addition, if a server experiences a burst of traffic, the burst is absorbed in event queues, providing *graceful degradation* by preserving the throughput of the server but temporarily increasing latency. By contrast, thread-per-task systems degrade in both throughput and latency if bursts are absorbed by additional threads.

3.1 Assumptions

If one DDS node cannot communicate with another, we assume it is because this other node has stopped executing (due to a planned shutdown or a crash); we assume that network partitions do not occur inside our cluster, and that DDS software components are fail-stop. The need for no network partitions is addressed by the high redundancy of our network, as previously mentioned. We have attempted to induce fail-stop behavior in our software by having it terminate its own execution if it encounters an unexpected condition, rather than attempting to gracefully recover from such a condition. These strong assumptions have been valid in practice; we have never experienced an unplanned network partition in our cluster, and our software has always behaved in a fail-stop manner. We further assume that software failures in the cluster are independent. We replicate all durable data at more than one place in the cluster, but we assume that at least one replica is active (has not failed) at all times. We also assume some degree of synchrony, in that processes take a bounded amount of time to execute tasks, and that messages take a bounded amount of time to be delivered.

We make several assumptions about the workload presented to our distributed hash tables. A table’s key space is the set of 64-bit integers; we

¹We do have a checkpoint mechanism (discussed later) that permits us to recover in the case that any of these cluster properties fail, however all state changes that happen after the last checkpoint will be lost should this occur.

assume that the population density over this space is even (i.e. the probability that a given key exists in the table is a function of the number of values in the table, but not of the particular key). We don’t assume that all keys are accessed equiprobably, but rather that the “working set” of hot keys is larger than the number of nodes in our cluster. We then assume that a partitioning strategy that maps fractions of the keyspace to cluster nodes based on the nodes’ relative processing speed will induce a balanced workload. Our current DDS design does not gracefully handle a small number of extreme hotspots (i.e., if a handful of keys receive most of the workload). If there are many such hotspots, however, then our partitioning strategy will probabilistically balance them across the cluster. Failure of these workload assumptions can result in load imbalances across the cluster, leading to a reduction in throughput.

Finally, we assume that tables are large and long lived. Hash table creations and destructions are relatively rare events: the common case is for hash tables to serve read, write, and remove operations.

4 Distributed Hash Tables: Architecture and Implementation

In this section, we present the architecture and implementation of a distributed hash table DDS. Figure 2 illustrates our hash table’s architecture, which consists of the following components:

Client: a client consists of service-specific software running on a client machine that communicates across the wide area with one of many service instances running in the cluster. The mechanism by which the client selects a service instance is beyond the scope of this work, but it typically involves DNS round robin [7], a service-specific protocol, or level 4 or level 7 load-balancing switches on the edge of the cluster. An example of a client is a web browser, in which case the service would be a web server. Note that clients are completely unaware of DDS’s: no part of the DDS system runs on a client.

Service: a service is a set of cooperating software processes, each of which we call a service instance. Service instances communicate with wide-area clients and perform some application-level function. Services may have soft state (state which may be lost and recomputed if necessary), but they rely on the hash table to manage all persistent state.

Hash table API: the hash table API is the boundary between a service instance and its “DDS library”. The API provides services with `put()`, `get()`, `remove()`, `create()`, and `destroy()` operations on hash tables. Each operation is atomic, and all services see the same coherent image of all exist-

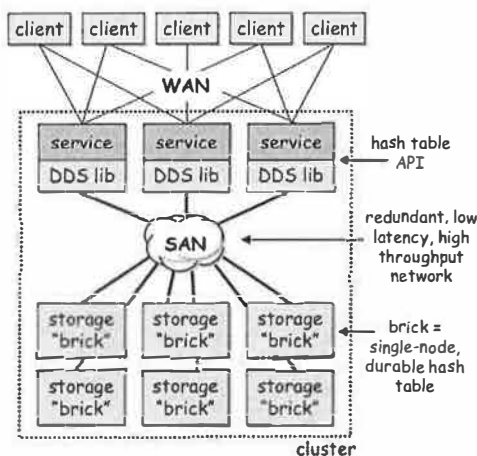


Figure 2: **Distributed hash table architecture:** each box in the diagram represents a software process. In the simplest case, each process runs on its own physical machine, however there is nothing preventing processes from sharing machines.

ing hash tables through this API. Hash table names are strings, hash table keys are 64 bit integers, and hash table values are opaque byte arrays; operations affect hash table values in their entirety.

DDS library: the DDS library is a Java class library that presents the hash table API to services. The library accepts hash table operations, and co-operates with the “bricks” to realize those operations. The library contains only soft state, including metadata about the cluster’s current configuration and the partitioning of data in the distributed hash tables across the “bricks”. The DDS library acts as the two-phase commit coordinator for state-changing operations on the distributed hash tables.

Brick: bricks are the only system components that manage durable data. Each brick manages a set of network-accessible single node hash tables. A brick consists of a buffer cache, a lock manager, a persistent chained hash table implementation, and network stubs and skeletons for remote communication. Typically, we run one brick per CPU in the cluster, and thus a 4-way SMP will house 4 bricks. Bricks may run on dedicated nodes, or they may share nodes with other components.

4.1 Partitioning, Replication, and Replica Consistency

A distributed hash table provides incremental scalability of throughput and data capacity as more nodes are added to the cluster. To achieve this, we horizontally partition tables to spread operations and data across bricks. Each brick thus stores some number of *partitions* of each table in the system, and when new nodes are added to the cluster, this parti-

tioning is altered so that data is spread onto the new node. Because of our workload assumptions (section 3.1), this horizontal partitioning evenly spreads both load and data across the cluster.

Given that the data in the hash table is spread across multiple nodes, if any of those nodes fail, then a portion of the hash table will become unavailable. For this reason, each partition in the hash table is replicated on more than one cluster node. The set of replicas for a partition form a *replica group*; all replicas in the group are kept strictly coherent with each other. Any replica can be used to service a `get()`, but all replicas must be updated during a `put()` or `remove()`. If a node fails, the data from its partitions is available on the surviving members of the partitions’ replica groups. Replica group membership is thus dynamic; when a node fails, all of its replicas are removed from their replica groups. When a node joins the cluster, it may be added to the replica groups of some partitions (such as in the case of recovery, described later).

To maintain consistency when state changing operations (`put()` and `remove()`) are issued against a partition, all replicas of that partition must be synchronously updated. We use an optimistic two-phase commit protocol to achieve consistency, with the DDS library serving as the commit coordinator and the replicas serving as the participants. If the DDS library crashes after *prepare* messages are sent, but before any *commit* messages are sent, the replicas will time out and abort the operation.

However, if the DDS library crashes after sending out any *commits*, then all replicas must commit. For the sake of availability, we do not rely on the DDS library to recover after a crash and issuing pending *commits*. Instead, replicas store short in-memory logs of recent state changing operations and their outcomes. If a replica times out while waiting for a *commit*, that replica communicates with all of its peers to find out if any have received a *commit* for that operation, and if so, the replica commits as well; if not, the replica aborts. Because all peers in the replica group that time out while waiting for a *commit* communicate with all other peers, if any receives a *commit*, then all will commit.

Any replica may abort during the first phase of the two-phase commit (e.g., if the replica cannot obtain a write lock on a key). If the DDS library receives any *abort* messages at the end of the first phase, it sends *aborts* to all replicas in the second phase. Replicas do not commit side-effects unless they receive a *commit* message in the second phase.

If a replica crashes during a two-phase commit, the DDS library simply removes it from its replica group and continues onward. Thus, all replica groups shrink over time; we rely on a recovery mech-

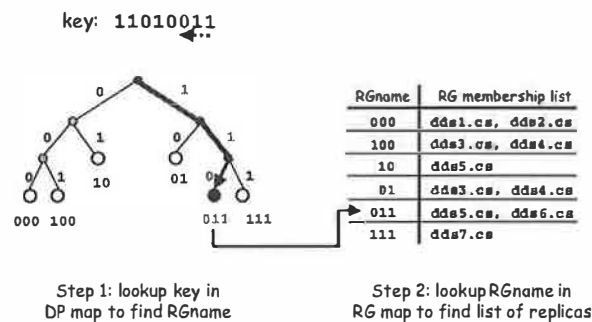


Figure 3: **Distributed hash table metadata maps:** this illustration highlights the steps taken to discover the set of replica groups which serve as the backing store for a specific hash table key. The key is used to traverse the DP map trie and retrieve the name of the key's replica group. The replica group name is then used looked up in the RG map to find the group's current membership.

anism (described later) for crashed replicas to rejoin the replica group. We made the significant optimization that the image of each replica must only be consistent through its brick's cache, rather than having a consistent on-disk image. This allows us to have a purely conflict-driven cache eviction policy, rather than having to force cache elements out to ensure on-disk consistency. An implication of this is that if all members of a replica group crash, that partition is lost. We assume nodes are independent failure boundaries (section 3.1); there must be no systematic software failure across nodes, and the cluster's power supply must be uninterruptible.

Our two-phase commit mechanism gives *atomic updates* to the hash table. It does not, however, give transactional updates. If a service wishes to update more than one element atomically, our DDS does not provide any help. Adding transactional support to our DDS infrastructure is a topic of future work, but this would require significant additional complexity such as distributed deadlock detection and undo/redo logs for recovery.

We do have a checkpoint mechanism in our distributed hash table that allows us to force the on-disk image of all partitions to be consistent; the disk images can then be backed up for disaster recovery. This checkpoint mechanism is extremely heavyweight, however; during the checkpointing of a hash table, no state-changing operations are allowed. We currently rely on system administrators to decide when to initiate checkpoints.

4.2 Metadata maps

To find the partition that manages a particular hash table key, and to determine the list of replicas in partitions' replica groups, the DDS libraries con-

sult two metadata maps that are replicated on each node of the cluster. Each hash table in the cluster has its own pair of metadata maps.

The first map is called the *data partitioning (DP) map*. Given a hash table key, the DP map returns the name of the key's partition. The DP map thus controls the horizontal partitioning of data across the bricks. As shown in figure 3, the DP map is a trie over hash table keys; to find a key's partition, key bits are used to walk down the trie, starting from the least significant key bit until a leaf node is found. As the cluster grows, the DP trie subdivides in a "split" operation. For example, partition 10 in the DP trie of figure 3 could split into partitions 010 and 110; when this happens, the keys in the old partition are shuffled across the two new partitions. The opposite of a split is a "merge"; if the cluster is shrunk, two partitions with a common parent in the trie can be merged into their parent. For example, partitions 000 and 100 in figure 3 could be merged into a single partition 00.

The second map is called the *replica group (RG) membership map*. Given a partition name, the RG map returns a list of bricks that are currently serving as replicas in the partition's replica group. The RG maps are dynamic: if a brick fails, it is removed from all RG maps that contain it. A brick joins a replica group after finishing recovery. An invariant that must be preserved is that the replica group membership maps for all partitions in the hash table must have at least one member.

The maps are replicated on each cluster node, in both the DDS libraries and the bricks. The maps must be kept consistent, otherwise operations may be applied to the wrong bricks. Instead of enforcing consistency synchronously, we allow the libraries' maps to drift out of date, but lazily update them when they are used to perform operations. The DDS library piggybacks hashes of the maps² on operations sent to bricks; if a brick detects that either map used is out of date, the brick fails the operation and returns a "repair" to the library. Thus, all maps become eventually consistent as they are used. Because of this mechanism, libraries can be restarted with out of date maps, and as the library gets used its maps become consistent.

To put() a key and value into a hash table, the DDS library servicing the operation consults its DP map to determine the correct partition for the key. It then looks up that partition name in its RG map to find the current set of bricks serving as replicas, and finally performs a two-phase commit across these replicas. To do a get() of a key, a similar process is used, except that the DDS library can

²It is important to use large enough of a hash to make the probability of collision negligible; we currently use 32 bits.

select any of the replicas listed in the RG map to service the read. We use the locality-aware request distribution (LARD) technique [14] to select a read replica—LARD further partitions keys across replicas, in effect aggregating their physical caches.

4.3 Recovery

If a brick fails, all replicas on it become unavailable. Rather than making these partitions unavailable, we remove the failed brick from all replica groups and allow operations to continue on the surviving replicas. When the failed brick recovers (or an alternative brick is selected to replace it), it must “catch up” to all of the operations it missed. In many RDBMS’s and file systems, recovery is a complex process that involves replaying logs, but in our system we use properties of clusters and our DDS design for vast simplifications.

Firstly, we allow our hash table to “say no”—bricks may return a failure for an operation, such as when a two-phase commit cannot obtain locks on all bricks (e.g., if two `puts()` to the same key are simultaneously issued), or when replica group memberships change during an operation. The freedom to say no greatly simplifies system logic, since we don’t worry about correctly handling operations in these rare situations. Instead, we rely on the DDS library (or, ultimately, the service and perhaps even the WAN client) to retry the operation. Secondly, we don’t allow any operation to finish unless all participating components agree on the metadata maps. If any component has an out-of-date map, operations fail until the maps are reconciled.

We make our partitions relatively small (~100MB), which means that we can transfer an entire partition over a fast system-area network (typically 100 Mb/s to 1 Gb/s) within 1 to 10 seconds. Thus, during recovery, we can incrementally copy entire partitions to the recovering node, obviating the need for the undo and redo logs that are typically maintained by databases for recovery. When a node initiates recovery, it grabs a write lease on one replica group member from the partition that it is joining; this write lease means that all state-changing operations on that partition will start to fail. Next, the recovering node copies the entire replica over the network. Then, it sends updates to the RG map to all other replicas in the group, which means that DDS libraries will start to lazily receive this update. Finally, it releases the write lock, which means that the previously failed operations will succeed on retry. The recovery of the partition is now complete, and the recovering node can begin recovery of other partitions as necessary.

There is an interesting choice of the rate at which partitions are transferred over the network

during recovery. If this rate is fast, then the involved bricks will suffer a loss in read throughput during the recovery. If this rate is slow, then the bricks won’t lose throughput, but the partition’s mean time to recovery will increase. We chose to recover as quickly as possible, since in a large cluster only a small fraction of the total throughput of the cluster will be affected by the recovery.

A similar technique is used for DP map split and merge operations, except that all replicas must be modified and both the RG and DP maps are updated at the end of the operation.

4.3.1 Convergence of Recovery

A challenge for fault-tolerant systems is to remain consistent in the face of repeated failures; our recovery scheme described above has this property. In steady state operation, all replicas in a group are kept perfectly consistent. During recovery, state changing operations fail (but only on the recovering partition), implying that surviving replicas remain consistent and recovering nodes have a stable image from which to recover. We also ensure that a recovering node only joins the replica group after it has successfully copied over the entire partition’s data but before it release its write lease. A remaining window of vulnerability in the system is if recovery takes longer than the write lease; if this seems imminent, the recovering node could aggressively renew its write lease, but we have not currently implemented this behavior.

If a recovering node crashes during recovery, its write lease will expire and the system will continue as normal. If the replica on which the lease was grabbed crashes, the recovering node must reinitiate recovery with another surviving member of the replica group. If all members of a replica group crash, data will be lost, as mentioned in Section 3.1.

4.4 Asynchrony

All components of the distributed hash table are built using an asynchronous, event-driven programming style. Each hash table layer is designed so that only a single thread ever executes in it at a time. This greatly simplified implementation by eliminating the need for data locks, and race conditions due to threads. Hash table layers are separated by FIFO queues, into which I/O completion events and I/O requests are placed. The FIFO discipline of these queues ensures fairness across requests, and the queues act as natural buffers that absorb bursts that exceed the system’s throughput capacity.

All interfaces in the system (including the DDS library APIs) are split-phase and asynchronous. This means that a hash table `get()` doesn’t block, but rather immediately returns with an identifier

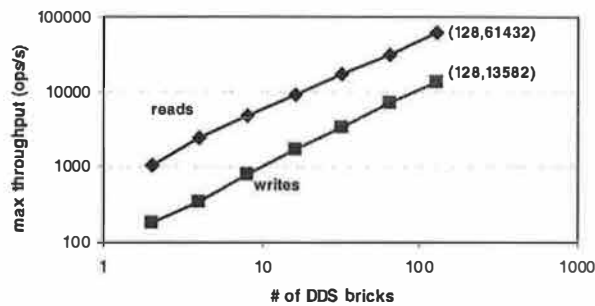


Figure 4: **Throughput scalability:** this benchmark shows the linear scaling of throughput as a function of the number of bricks serving in a distributed hash table; note that both axis have logarithmic scales. As we added more bricks to the DDS, we increased the number of clients using the DDS until throughput saturated.

that can be matched up with a completion event that is delivered to a caller-specified upcall handler. This upcall handler can be application code, or it can be a queue that is polled or blocked upon.

5 Performance

In this section, we present performance benchmarks of the distributed hash table implementation that were gathered on a cluster of 28 2-way SMPs and 38 4-way SMPs (a total of 208 500 MHz Pentium CPUs). Each 2-way SMP has 500 MB of RAM, and each 4-way SMP has 1 GB. All are connected with either 100 Mb/s switched Ethernet (2-way SMPs) or 1 Gb/s switched Ethernet (4-way SMPs). The benchmarks are run using Sun's JDK 1.1.7v3, using the OpenJIT 1.1.7 JIT compiler and "green" (user-level) threads on top of Linux v2.2.5.

When running our benchmarks, we evenly spread hash table bricks amongst 4-way and 2-way SMPs, running at most one brick node per CPU in the cluster. Thus, 4-way SMPs would have at most 4 brick processes running on them, while 2-way SMPs would have at most 2. We also made use of these cluster nodes as load generators; because of this, we were only able to gather performance numbers to a maximum of a 128 brick distributed hash table, as we needed the remaining 80 CPUs to generate enough load to saturate such a large table.

5.1 In-Core Benchmarks

Our first set of benchmarks tested the in-core performance of the distributed hash table. By limiting the working set of keys that we requested to a size that fits in the aggregate physical memory of the bricks, this set of benchmarks investigates the overhead and throughput of the distributed hash table code independently of disk performance.

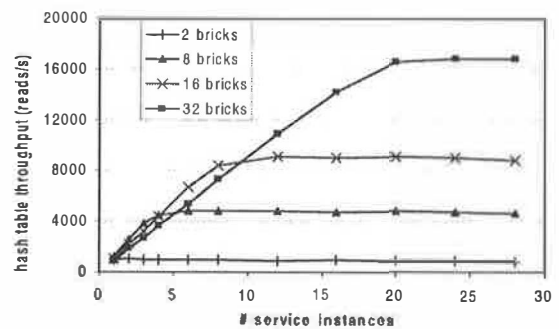


Figure 5: **Graceful degradation of reads:** this graph demonstrates that the read throughput from a distributed hash table remains constant even if the offered load exceeds the capacity of the hash table.

5.1.1 Throughput Scalability

This benchmark demonstrates that hash table throughput scales linearly with the number of bricks. The benchmark consists of several services that each maintain a pipeline of 100 operations (either `gets()` or `puts()`) to a single distributed hash table. We varied the number of bricks in the hash table; for each configuration, we slowly increased the number of services and measured the completion throughput flowing from the bricks. All configurations had 2 replicas per replica group, and each benchmark iteration consisted of reads or writes of 150-byte values. The benchmark was closed-loop: a new operation was immediately issued with a random key for each completed operation.

Figure 4 shows the maximum throughput sustained by the distributed hash table as a function of the number of bricks. Throughput scales linearly up to 128 bricks; we didn't have enough processors to scale the benchmark further. The read throughput achieved with 128 bricks is 61,432 reads per second (5.3 billion per day), and the write throughput with 128 bricks is 13,582 writes per second (1.2 billion per day); this performance is adequate to serve the hit rates of most popular web sites on the Internet.

5.1.2 Graceful Degradation for Reads

Bursts of traffic are a common phenomenon for all Internet services. If a traffic burst exceeds the service's capacity, the service should have the property of "graceful degradation": the throughput of the service should remain constant, with the excess traffic either being rejected or absorbed in buffers and served with higher latency. Figure 5 shows the throughput of a distributed hash table as a function of the number of simultaneous read requests issued to it; each service instance has a closed-loop pipeline of 100 operations. Each line on the graph represents a different number of bricks serving the

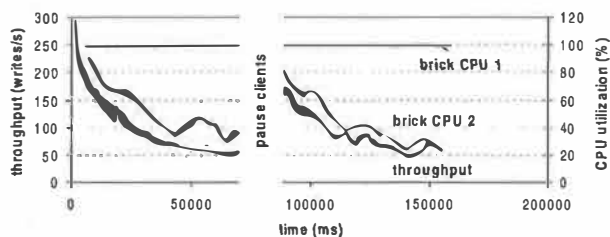


Figure 6: **Write imbalance leading to ungraceful degradation:** the bottom curve shows the throughput of a two-brick partition under overload, and the top two curves show the CPU utilization of those bricks. One brick is saturated, the other becomes only 30% busy.

hash table. Each configuration is seen to eventually reach a maximum throughput as its bricks saturate. This maximum throughput is successfully sustained even as additional traffic is offered. The overload traffic is absorbed in the FIFO event queues of the bricks; all tasks are processed, but they experience higher latency as the queues drain from the burst.

5.1.3 Ungraceful Degradation for Writes

An unfortunate performance anomaly emerged when benchmarking `put()` throughput. As the offered load approached the maximum capacity of the hash table bricks, the total write throughput suddenly began to drop. On closer examination, we discovered that most of the bricks in the hash table were unloaded, but one brick in the hash table was completely saturated and had become the bottleneck in the closed-loop benchmark.

Figure 6 illustrates this imbalance. To generate it, we issued `puts()` to a hash table with a single partition and two replicas in its replica group. Each `put()` operation caused a two-phase commit across both replicas, and thus each replica saw the same set of network messages and performed the same computation (but perhaps in slightly different orders). We expected both replicas to perform identically, but instead one replica became more and more idle, and the throughput of the hash table dropped to match the CPU utilization of this idle replica.

Investigation showed that the busy replica was spending a significant amount of time in garbage collection. As more live objects populated that replica's heap, more time needed to be spent garbage collecting to reclaim a fixed amount of heap space, as more objects would be examined before a free object was discovered. Random fluctuations in arrival rates and garbage collection caused one replica to spend more time garbage collecting than the other. This replica became the system bottleneck, and more operations piled up in its queues, further amplifying this imbalance. Write traffic particularly ex-

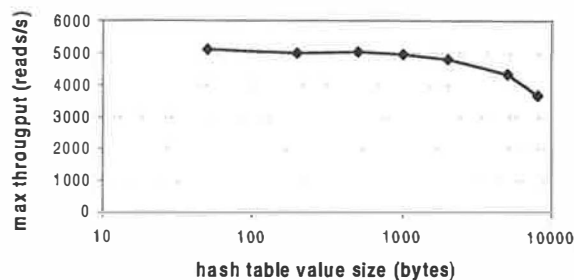


Figure 7: **Throughput vs. read size** the X axis shows the size of values read from the hash table, and the Y axis shows the maximum throughput sustained by an 8 brick hash table serving these values.

acerbated the situation, as objects created by the “prepare” phase must wait for at least one network round-trip time before a commit or abort command in the second phase is received. The number of live objects in each bricks' heap is thus proportional to the bandwidth-delay product of hash table `put()` operations. For read traffic, there is only one phase, and thus objects can be garbage collected immediately after read requests are satisfied.

We experimented with many JDKs, but consistently saw this effect. Some JDKs (such as JDK 1.2.2 on Linux 2.2.5) developed this imbalance for read traffic as well as write traffic. This sort of performance imbalance is fundamental to any system that doesn't perform admission control; if the task arrival rate temporarily exceeds the system's ability to handle them, then tasks will begin to pile up in the system. Because systems have finite resources, this inevitably causes performance degradation (thrashing). In our system, this degradation first materialized due to garbage collection. In other systems, this might happen due to virtual memory thrashing, to pick an example. We are currently exploring using admission control (at either the bricks or the hash table libraries) or early discard from bricks' queues to keep the bricks within their operational range, ameliorating this imbalance.

5.1.4 Throughput Bottlenecks

In figure 7, we varied the size of elements that we read out of an 8 brick hash table. Throughput was flat from 50 bytes through 1000 bytes, but then began to degrade. From this we deduced that per-operation overhead (such as object creation, garbage collection, and system call overhead) saturated the bricks' CPUs for elements smaller than 1000 bytes, and per-byte overhead (byte array copies, either in the TCP stack or in the JVM) saturated the bricks' CPUs for elements greater than 1000 bytes. At 8000 bytes, the throughput in and out of each 2-way SMP (running 2 bricks) was 60 Mb/s. For larger sized

hash table values, the 100 Mb/s switched network became the throughput bottleneck.

5.2 Out-of-core Benchmarks

Our next set of benchmarks tested performance for workloads that do not fit in the aggregate physical memory of the bricks. These benchmarks stress the single-node hash table's disk interaction, as well as the performance of the distributed hash table.

5.2.1 A Terabyte DDS

To test how well the distributed hash table scales in terms of data capacity, we populated a hash table with 1.28 terabytes of 8KB data elements. To do this, we created a table with 512 partitions in its DP map, but with only 1 replica per replica group (i.e., the table would not withstand node failures). We spread the 512 partitions across 128 brick nodes, and ran 2 bricks per node in the cluster. Each brick stored its data on a dedicated 12GB disk (all cluster nodes have 2 of these disks). The bricks each used 10GB worth of disk capacity, resulting in 1.28TB of data stored in the table.

To populate the 1.28TB hash table, we designed bulk loaders that generated writes to keys in an order that was carefully chosen to result in sequential disk writes. These bulk loaders understood the partitioning in the DP map and implementation details about the single-node tables' hash functions (which map keys to disk blocks). Using these loaders, it took 130 minutes to fill the table with 1.28 terabytes of data, achieving a total write throughput of 22,015 operations/s, or 1.4 MB/s per disk.

Comparatively, the in-core throughput benchmark presented in Section 5.1.1 obtained 13,582 operations/s for a 128 brick table, but that benchmark was configured with 2 replicas per replica group. Eliminating this replication would double the throughput of the in-core benchmark, resulting in a 27,164 operations/s. The bulk loading of the 1.28TB hash table was therefore only marginally slower in terms of the throughput sustained by each replica than the in-core benchmarks, which means that disk throughput was not the bottleneck.

5.2.2 Random Write and Read Throughput

However, we believe it is unrealistic and undesirable for hash table clients to have knowledge of the DP map and single-node tables' hash functions. We ran a second set of throughput benchmarks on another 1.28TB hash table, but populated it with random keys. With this workload, the table took 319 minutes to populate, resulting in a total write throughput of 8,985 operations/s, or 0.57 MB/s per

disk. We similarly sustained a read throughput of 14,459 operations/s, or 0.93 MB/s per disk.³

This throughput is substantially lower than the throughput obtained during the in-core benchmarks because the random workload generated results in random read and write traffic to each disk. In fact, for this random workload, every `read()` issued to the distributed hash table results in a request for a random disk block from a disk. All disk traffic is seek dominated, and disk seeks become the overall bottleneck of the system.

We expect that there will be significant locality in DDS requests generated by Internet services, and given workloads with high locality, the DDS should perform nearly as well as the in-core benchmark results. However, it might be possible to significantly improve the write performance of traffic with little locality by using disk layout techniques similar to those of log-structured file systems [29]; we have not explored this possibility as of yet.

5.3 Availability and Recovery

To demonstrate availability in the face of node failures and the ability for the bricks to recover after a failure, we repeated the read benchmark with a hash table of 150 byte elements. The table was configured with a single 100MB partition and three replicas in that partition's replica group. Figure 8 shows the throughput of the hash table over time as we induced a fault in one of the replica bricks and later initiated its recovery. During recovery, the rate at which the recovered partition is copied was 12 MB/s, which is maximum sequential write bandwidth we could obtain from the bricks' disks.

At point (1), all three bricks were operational and the throughput sustained by the hash table was 450 operations per second. At point (2), one of the three bricks was killed. Performance immediately dropped to 300 operations per second, two-thirds of the original capacity. Fault detection was immediate: client libraries experienced broken transport connections that could not be reestablished. The performance overhead of the replica group map updates could not be observed. At point (3), recovery was initiated, and recovery completed at point (4). Between points (3) and (4), there was no noticeable performance overhead of recovery; this is because there was ample excess bandwidth on the network, and the CPU overhead of transferring the partition during recovery was negligible. It should be noted that between points (3) and (4), the recov-

³Write throughput is less than read throughput because a hash bucket must be read before it can be written, in case there is already data stored in that bucket that must be preserved. There is therefore an additional read for every write, nearly halving the effective throughput for DDS writes.

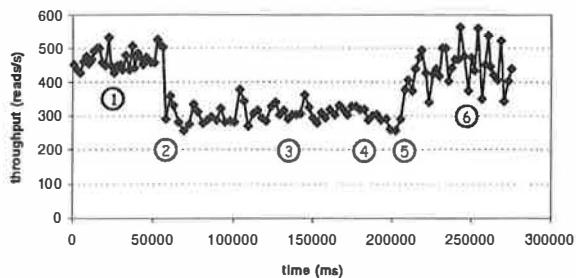


Figure 8: **Availability and Recovery:** this benchmark shows the read throughput of a 3-brick hash table as a deliberate single-node fault is induced, and afterwards as recovery is performed.

ering partition is not available for writes, because of the write lease grabbed during recovery. This partition is available for reads, however.

After recovery completed, performance briefly dropped at point (5). This degradation is due to the buffer cache warming on the recovered node. Once the cache became warm, performance resumed to the original 450 operations/s at point (6). An interesting anomaly at point (6) is the presence of noticeable oscillations in throughput; these were traced to garbage collection triggered by the “extra” activity of recovery. When we repeated our measurements, we would occasionally see this oscillation at other times besides immediately post-recovery. This sort of performance unpredictability due to garbage collection seems to be a pervasive problem; a better garbage collector or admission control might ameliorate this, but we haven’t yet explored this.

6 Example Services

We have implemented a number of interesting services using our distributed hash table. The services’ implementation was greatly simplified by using the DDS, and they trivially scaled by adding more service instances. An aspect of scalability not covered by using the hash table was the routing and load balancing of WAN client requests across service instances, but this is beyond the scope of this work.

Sanctio: Sanctio is an instant messaging gateway that provides protocol translation between popular instant messaging protocols (such as Mirabilis’ ICQ and AOL’s AIM), conventional email, and voice messaging over cellular telephones. Sanctio is a middleman between these protocols, routing and translating messages between the networks. In addition to protocol translation, Sanctio also can transform the message content. We have built a “web scraper” that allows us to compose AltaVista’s BabelFish natural language translation service with Sanctio. We can thus perform language translation (e.g., English to French) as well as protocol translation; a

Spanish speaking ICQ user can send a message to an English speaking AIM user, with Sanctio providing both language and protocol translation.

A user may be reached on a number of different addresses, one for each of the networks that Sanctio can communicate with. The Sanctio service must therefore keep a large table of bindings between users and their current transport addresses on these networks; we used the distributed hash table for this purpose. The expected workload on the DDS includes significant write traffic generated when users change networks or log in and out of a network. The data in the table must be kept consistent, otherwise messages will be routed to the wrong address.

Sanctio took 1 person-month to develop, most of which was spent authoring the protocol translation code. The code that interacts with the distributed hash table took less than a day to write.

Web server: we have implemented a scalable web server using the distributed hash table. The server speaks HTTP to web clients, hashes requested URLs into 64 bit keys, and requests those keys from the hash table. The server takes advantage of the event-driven, queue-centric programming style to introduce CGI-like behavior by interposing on the URL resolution path. This web server was written in 900 lines of Java, 750 of which deals with HTTP parsing and URL resolution, and only 50 of which deals with interacting with the hash table DDS.

Others: We have built many other services as part of the Ninja project⁴. The “Parallelisms” service recommends related sites to user-specified URLs by looking up ontological entries in an inversion of the Yahoo web directory. We built a collaborative filtering engine for a digital music jukebox service [16]; this engine stores users’ music preferences in a distributed hash table. We have also implemented a private key store and a composable user preference service, both of which use the distributed hash table for persistent state management.

7 Discussion

Our experience with the distributed hash table implementation has taught us many lessons about using it as a storage platform for scalable services. The hash table was a resounding success in simplifying the construction of interesting services, and these services inherited the scalability, availability, and data consistency of the hash table. Exploiting properties of clusters also proved to be remarkably useful. In our experience, most of the assumptions that we made regarding properties of a clusters and component failures (specifically the fail-stop behav-

⁴<http://ninja.cs.berkeley.edu/>

ior of our software and the probabilistic lack of network partitions in the cluster) were valid in practice.

One of our assumptions was initially problematic: we observed a case where there was a systematic failure of all replica group members inside a single replica group. This failure was caused by a software bug that enabled service instances to deterministically crash remote bricks by inducing a null pointer exception in the JVM. After fixing the associated bug in the brick, this situation never again arose. However, it serves as a reminder that systematic software bugs can in practice bring down the entire cluster at once. Careful software engineering and a good quality assurance cycle can help to ameliorate this failure mode, but we believe that this issue is fundamental to all systems that promise both availability and consistency.

As we scaled our distributed hash table, we noticed scaling bottlenecks that weren't associated with our own software. At 128 bricks, we approached the point at which the 100 Mb/s Ethernet switches would saturate; upgrading to 1 Gb/s switches throughout the cluster would delay this saturation. We also noticed that the combination of our JVM's user-level threads and the Linux kernel began to induced poor scaling behavior as each node in the cluster opened up a reliable TCP connection to all other nodes in the cluster. The brick processes began to saturate due to a flood of signals from the kernel to the user-level thread scheduler associated with TCP connections with data waiting to be read.

7.1 Java as a Service Platform

We found that Java was an adequate platform from which to build a scalable, high performance subsystem. However, we ran into a number of serious issues with the Java language and runtime. The garbage collector of all JVMs that we experimented with inevitably became the performance bottleneck of the bricks and also a source of throughput and latency variation. Whenever the garbage collector became active, it had a serious impact on all other system activity, and unfortunately, current JVMs do not provide adequate interfaces to allow systems to control garbage collection behavior.

The type safety and array bounds checking features of Java vastly accelerated our software engineering process, and helped us to write stable, clean code. However, these features got in the way of code efficiency, especially when dealing with multiple layers of a system each of which wraps some array of data with layer-specific metadata. We often found ourselves performing copies of regions of byte arrays in order to maintain clean interfaces to data regions, whereas in a C implementation it would be more natural to exploit pointers into malloc'ed memory

regions to the same effect without needing copies.

Java lacks asynchronous I/O primitives, which necessitated the use of a thread pool at the lowest-layer of the system. This is much more efficient than a thread-per-task system, as the number of threads in our system is equal to the number of outstanding I/O requests rather than the number of tasks. Nonetheless, it introduced performance overhead and scaling problems, since the number of TCP connections per brick increases with the cluster size. We are working on introducing high-throughput asynchronous I/O completion mechanisms into the JVM using the JNI native interface.

7.2 Future Work

We plan on investigating more interesting data-parallel operations on a DDS (such as an iterator, or the Lisp `maplist()` operator). We also plan on building other distributed data structures, including a B-tree and an administrative log. In doing so, we hope to reuse many of the components of the hash table, such as the brick storage layer, the RG map infrastructure, and the two-phase commit code. We would like to explore caching in the DDS libraries (we currently rely on services to build their own application-level caches). We are also exploring adding other single-element operations to the hash table, such as `testandset()`, in order to provide locks and leases to services that may have many service instances competing to write to the same hash table element.

8 Related Work

✱ Litwin et al.'s scalable, distributed data structures (SDDS) such as *RP** [22, 26] helped to motivate our own work. *RP** focuses on algorithmic properties, while we focused on the systems issues of implementing an SDDS that satisfies the concurrency, availability, and incremental scalability needs of Internet services.

Our work has a great deal in common with database research. The problems of partitioning and replicating data across shared-nothing multi-computers has been studied extensively in the distributed and parallel database communities [10, 17, 25]. We use mechanisms such as horizontal partitioning and two-phase commits, but we do not need an SQL parser or a query optimization layer since we have no general-purpose queries in our system.

We also have much in common with distributed and parallel file systems [3, 23, 31, 33]. A DDS presents a higher-level interface than a typical file system, and DDS operations are data-structure specific and atomically affect entire elements. Our research has focused on scalability, availability, and

consistency under high throughput, highly concurrent traffic, which is a different focus than file systems. Our work is most similar to Petal [24], in that a Petal distributed virtual disk can be thought of as a simple hash table with fixed sized elements. Our hash tables have variable sized elements, an additional name space (the set of hash tables), and focus on Internet service workloads and properties as opposed to file system workloads and properties.

The CMU network attached secure disk (NASD) architecture [11] explores variable-sized object interfaces as an abstraction to allow storage subsystems to optimize disk layout. This is similar to our own data structure interface, which is deliberately higher-level than the block or file interfaces of Petal and parallel or distributed file systems.

Distributed object stores [13] attempt to transparently adding persistence to distributed object systems. The persistence of (typed) objects is typically determined by reachability through the transitive closure of object references, and the removal of objects is handled by garbage collection. A DDS has no notion of pointers or object typing, and applications must explicitly use API operations to store and retrieve elements from a DDS. Distributed object stores are often built with the wide-area in mind, and thus do not focus on the scalability, availability, and high throughput requirements of cluster-based Internet services.

Many projects have explored the use of clusters of workstations as a general-purpose platform for building Internet services [1, 4, 15]. To date, these platforms rely on file systems or databases for persistent state management; our DDS's are meant to augment such platforms with a state management platform that is better suited to the needs of Internet services. The Porcupine project [30] includes a storage platform built specifically for the needs of a cluster-based scalable mail server, but they are attempting to generalize their storage platform for arbitrary service construction.

There have been many projects that explored wide-area replicated, distributed services [9, 27]. Unlike clusters, wide-area systems must deal with heterogeneity, network partitions, untrusted peers, high latency and low throughput networks, and multiple administrative domains. Because of these differences, wide-area distributed systems tend to have relaxed consistency semantics and low update rates. However, if designed correctly, they can scale up enormously.

9 Conclusions

This paper presents a new persistent data management layer that enhances the ability of clusters to

support Internet services. This self-managing layer, called a distributed data structure (DDS), fills in an important gap in current cluster platforms by providing a data storage platform specifically tuned for services' workloads and for the cluster environment.

This paper focused on the design and implementation of a distributed hash table DDS, empirically demonstrating that it has many properties necessary for Internet services (incremental scaling of throughput and data capacity, fault tolerance and high availability, high concurrency, and consistency and durability of data). These properties were achieved by carefully designing the partitioning, replication, and recovery techniques in the hash table implementation to exploit features of cluster environments (such as a low-latency network with a lack of network partitions). By doing so, we have "right-sized" the DDS to the problem of persistent data management for Internet services.

The hash table DDS simplifies Internet service construction by decoupling service-specific logic from the complexities of persistent state management, and by allowing services to inherit the necessary service properties from the DDS rather than having to implement the properties themselves.

Acknowledgements

We are very grateful to Eric Anderson, Rob von Behren, Nikita Borisov, Mike Chen, Armando Fox, Jim Gray, Ramki Gummadi, Drew Roselli, Geoff Voelker, the anonymous referees, and our shepherd Bill Weihl for their very helpful suggestions that greatly improved the quality of this paper. We would also like to thank Eric Fraser, Phil Buonadonna, and Brent Chun for their help in giving us access to the Berkeley Millennium cluster for our performance benchmarks.

References

- [1] E. Amir, S. McCanne, and R. Katz. An Active Service Framework and its Application to Real-Time Multimedia Transcoding. In *Proceedings of ACM SIGCOMM '98*, pages 178–189, Oct 1998.
- [2] T. E. Anderson, D. E. Culler, and D. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, 12(1):54–64, Feb 1995.
- [3] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless Network File Systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Dec 1995.
- [4] D. Andresen, T. Yang, O. Egecioglu, O. H. Ibarra, and T. R. Smith. Scalability Issues for High Performance Digital Libraries on the World Wide Web. In *Proceedings of IEEE ADL '96*, Washington D.C., May 1996.

- [5] G. Banga, J. C. Mogul, and P. Druschel. A Scalable and Explicit Event Delivery Mechanism for UNIX. In *Proceedings of the USENIX 1999 Annual Technical Conference*, Monterey, CA, Jun 1999.
- [6] BEA Systems. BEA WebLogic Application Servers. <http://www.bea.com/products/weblogic/>.
- [7] T. Brisco. RFC 1764: DNS Support for Load Balancing, Apr 1995.
- [8] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 Usenix Annual Technical Conference*, Jan 1996.
- [9] A.D. Birrell et al. Grapevine: An Exercise in Distributed Computing. *Communications of the ACM*, 25(4):3–23, Feb 1984.
- [10] D. DeWitt et al. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), Mar 1990.
- [11] G. A. Gibson et al. A Cost-Effective, High-Bandwidth Storage Architecture. In *ASPLOS-VIII*, San Jose, California, 1998.
- [12] J. H. Howard et al. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1), Feb 1988.
- [13] P. Ferreira et al. PerDiS: Design, Implementation, and Use of a PERsistent DIstributed Store. In *Recent Advances in Distributed Systems*, volume 1752 of *Lecture Notes in Computer Science*, chapter 18, pages 427–452. Springer-Verlag, Feb 2000.
- [14] V. S. Pai et al. Locality-Aware Request Distribution in Cluster-Based Network Servers. In *ASPLOS-VIII*, San Jose, CA, Oct 1998.
- [15] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St.-Malo, France, Oct 1997.
- [16] I. Goldberg, S. D. Gribble, D. Wagner, and E. A. Brewer. The Ninja Jukebox. In *The 2nd USENIX Symposium on Internet Technologies and Systems*, Boulder, CO, Oct 1999.
- [17] G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *ACM SIGMOD Conference on the Management of Data*, Atlantic City, NJ, May 1990.
- [18] Jim Gray. The Transaction Concept: Virtues and Limitations. In *Proceedings of VLDB*, Cannes, France, September 1981.
- [19] S. D. Gribble and E. A. Brewer. System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace. In *Proceedings of the 1997 USENIX Symposium on Internet Technologies and Systems (USITS 97)*, Monterey, CA, Dec 1997.
- [20] J. C. Hu, I. Pyarali, and D. C. Schmidt. Applying the Proactor Pattern to High-Performance Web Servers. In *Proceedings of the 10th International Conference on Parallel and Distributed Computing and Systems*, Oct 1998.
- [21] A. Iyengar, J. Challenger, D. Dias, and P. Dantzic. High-Performance Web Site Design Techniques. *IEEE Internet Computing*, 4(2), Mar 2000.
- [22] J. S. Karlsson, W. Litwin, and T. Risch. LH*LH: A Scalable High Performance Data Structure for Switched Multicomputers. In *Proceedings of the 5th International Conference on Extending Database Technology*, pages 573–591, Avignon, France, Mar 1996.
- [23] O. Krieger and M. Stumm. HFS: A Flexible File System for Large-Scale Multiprocessors. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 6–14, Hanover, NH, Jun 1993.
- [24] E. K. Lee and C. A. Thekkath. Petal: Distributed Virtual Disks. In *ASPLOS-VII*, Cambridge, MA, 1996.
- [25] B. G. Lindsay. A Retrospective of R*: A Distributed Database Management System. *Proceedings of the IEEE*, 75(5):668–673, May 1987.
- [26] W. Litwin, M. Neimat, and D. A. Schneider. RP*: A Family of Order Preserving Scalable Distributed Data Structures. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 342–353, Santiago, Chile, 1994.
- [27] P. V. Mockapetris and K. J. Dunlap. Development of the Domain Name System. In *ACM SIGCOMM Computer Communication Review*, 1988.
- [28] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proceedings of the 1999 Annual Usenix Technical Conference*, Jun 1999.
- [29] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, 1991.
- [30] Y. Saito, B. Bershad, and H. Levy. Manageability, Availability and Performance in Porcupine: a Highly Scalable, Cluster-based Mail Service. In *Proceedings of the 17th Symposium on Operating System Principles*, Kiawah Island, SC, Dec 1999.
- [31] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the USENIX 1985 Summer Conference*, El Cerrito, CA, Jun 1985.
- [32] J. Song, E. Levy, A. Iyengar, and D. Dias. Design Alternatives for Scalable Web Server Accelerators. In *Proceedings of the 2000 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2000)*, Austin, TX, Apr 2000.
- [33] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A Scalable Distributed File System. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St.-Malo, France, Oct 1997.

Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java

Godmar Back, Wilson C. Hsieh, Jay Lepreau
School of Computing
University of Utah

Abstract

Single-language runtime systems, in the form of Java virtual machines, are widely deployed platforms for executing untrusted mobile code. These runtimes provide some of the features that operating systems provide: inter-application memory protection and basic system services. They do not, however, provide the ability to isolate applications from each other, or limit their resource consumption. This paper describes KaffeOS, a Java runtime system that provides these features. The KaffeOS architecture takes many lessons from operating system design, such as the use of a user/kernel boundary, and employs garbage collection techniques, such as write barriers.

The KaffeOS architecture supports the OS abstraction of a *process* in a Java virtual machine. Each process executes as if it were run in its own virtual machine, including separate garbage collection of its own heap. The difficulty in designing KaffeOS lay in balancing the goals of isolation and resource management against the goal of allowing direct sharing of objects. Overall, KaffeOS is no more than 11% slower than the freely available JVM on which it is based, which is an acceptable penalty for the safety that it provides. Because of its implementation base, KaffeOS is substantially slower than commercial JVMs for trusted code, but it clearly outperforms those JVMs in the presence of denial-of-service attacks or misbehaving code.

1 Introduction

The need to support the safe execution of untrusted programs in runtime systems for type-safe languages has become clear. Language runtimes are being used in

many environments for executing untrusted code: for example, applets, servlets, active packets [41], database queries [15], and kernel extensions [6]. Current systems (such as Java) provide *memory protection* through the enforcement of type safety and *secure system services* through a number of mechanisms, including namespace and access control. Unfortunately, malicious or buggy applications can deny service to other applications. For example, a Java applet can generate excessive amounts of garbage and cause a Web browser to spend all of its time collecting it.

To support the execution of untrusted code, type-safe language runtimes need to provide a mechanism to isolate and manage the resources of applications, analogous to that provided by operating systems. Although other resource management abstractions exist [4], the classic OS *process* abstraction is appropriate. A process is the basic unit of resource ownership and control; it provides isolation between applications. On a traditional operating system, untrusted code can be forked in its own process; CPU and memory limits can be placed on the process; and the process can be killed if it is uncooperative.

A number of approaches to isolating applications in Java have been developed by others over the last few years. An *applet context* [9] is an example of an application-specific approach. It provides a separate namespace and a separate set of execution permissions for untrusted applets. Applet contexts do not support resource management, and cannot defend against denial-of-service attacks. In addition, they are not general: applet contexts are specific to applets, and cannot be used easily in other environments.

Several general-purpose models for isolating applications in Java do exist, such as the J-Kernel [23] or Echidna [21]. However, these solutions superimpose an operating system kernel abstraction on Java without changing the underlying virtual machine. As a result, it is impossible in those systems to account for resources spent on behalf of a given application: for example, CPU time spent while garbage collecting a process's heap.

An alternative approach to separate different applications is to give each one its own virtual machine, and run each virtual machine in a different process on an un-

This research was largely supported by the Defense Advanced Research Projects Agency, monitored by the Air Force Research Laboratory, Rome Research Site, USAF, under agreements F30602-96-2-0269 and F30602-99-1-0503. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon.

Contact information: {gback,wilson,lepreau}@cs.utah.edu. School of Computing, 50 S. Central Campus Drive, Room 3190, University of Utah, SLC, UT 84112-9205. <http://www.cs.utah.edu/flux/>

derlying OS [25, 29]. For instance, most operating systems can limit a process's heap size or CPU consumption. Such mechanisms could be used to directly limit an entire VM's resource consumption, but they depend on underlying operating system support.

Designing JVMs to support multiple processes is a superior approach. First, it reduces per-application overhead. For example, applications on KaffeOS can share classes in the same way that an OS allows applications to share libraries. Second, communication between processes can be more efficient in one VM, since objects can be shared directly. (One of the reasons for using type-safe language technology in systems such as SPIN [6] was to reduce the cost of IPC; we want to keep that goal.) Third, embedding a JVM in another application, such as a web server or web browser, is difficult (or impossible) if the JVM relies on an operating system to isolate different activities. Fourth, embedded or portable devices may not provide OS or hardware support for managing processes. Finally, a single JVM uses less energy than multiple JVM's on portable devices [19].

Our work consists of supporting processes in a modern type-safe language, Java. Our solution, KaffeOS, adds a process model to Java that allows a JVM to run multiple untrusted programs safely, and still supports the direct sharing of resources between programs. The difficulty in designing KaffeOS lay in balancing conflicting goals: process isolation and resource management versus direct sharing of objects between processes.

A KaffeOS process is a general-purpose mechanism that can easily be used in multiple application domains. For instance, KaffeOS could be used in a browser to support multiple applets, within a server to support multiple servlets, or even to provide a standalone "Java OS" on bare hardware. We have structured our abstractions and APIs so that they are as broadly applicable as possible, much as the OS process abstraction is. Because the KaffeOS architecture is designed to support processes, we have taken lessons from the design of traditional operating systems, such as the use of a user/kernel boundary.

Our design makes KaffeOS's isolation and resource control mechanisms comprehensive. We focus on the management of CPU time and memory, although we plan to address other resources such as network bandwidth. The runtime system is able to account for and control all of the CPU and memory resources consumed on behalf of any process. We have dealt with these issues by structuring the KaffeOS virtual machine so that it separates the resources used by different processes as much as possible.

To summarize, this paper makes the following contributions:

- We describe how lessons from building traditional operating systems can and should be used to structure runtime systems for type-safe languages.
- We describe how software mechanisms in the compiler and runtime can be used to implement isolation and resource management in a Java virtual machine.
- We describe the design and implementation of KaffeOS. KaffeOS implements our process model in Java, which isolates applications from each other, provides resource management mechanisms for them, and also lets them share resources directly.
- We show that the performance penalty for using KaffeOS is reasonable, compared to the freely available JVM on which it is based. Even though, due to that implementation base, KaffeOS is substantially slower than commercial JVMs on standard benchmarks, it outperforms those JVMs in the presence of uncooperative code.

Sections 2 and 3 describe and discuss the design and implementation of KaffeOS. Section 4 provides some performance measurements of KaffeOS, and compares its performance with that of some commercial Java virtual machines. Section 5 describes related work in more detail, and Section 6 summarizes our conclusions and results.

2 Design Principles

The following principles drove our design of KaffeOS, in decreasing order of importance:

- **Process separation.** We provide the "classical" property of a process: each process is given the illusion of having the whole virtual machine to itself.
- **Safe termination of processes.** Processes may terminate abruptly due to either an internal error or an external event. In both cases, we ensure that the integrity of other processes and the system itself is not violated.
- **Direct sharing between processes.** Processes can directly share objects in order to communicate with each other.
- **Precise memory and CPU accounting.** The memory and CPU time spent on almost all activities can be attributed to the application on whose behalf it was expended.

- **Full reclamation of memory.** When a process is terminated, its memory must be fully reclaimed. In a language-based system, memory cannot be revoked by unmapping pages: it must be garbage-collected. We restrict a process's heap writes to avoid uncollectable memory in the presence of direct object sharing.
- **Hierarchical memory management.** Memory allocation can be managed in a hierarchy, which provides a simple model for controlling processes.

The interaction between these design principles is complex. For expository purposes, we discuss these principles in a slightly different order in the remainder of this section.

Process separation. A process cannot accidentally or intentionally access another process' data, because each process has its own heap. A heap constitutes of a memory pool managed by an allocator and a garbage collector. Each process is given its own name space for its objects and classes, as well. Type safety provides memory protection, so that a process cannot access other process's objects.

To ensure process separation, an untrusted process is not allowed to hold onto system-level resources indefinitely. For instance, global kernel locks are not directly accessible to user processes. Violations of this restriction are instances of bad system design. Similarly, faults in one process must not impact progress in other processes.

Safe termination of processes. KaffeOS is structured such that critical parts of the system cannot be damaged when a process is terminated. For example, a process is not allowed to terminate when it is holding a lock on a system resource.

We divide KaffeOS into user and kernel parts [2], an important distinction used in operating system design. A user/kernel distinction is necessary to maintain system integrity in the presence of process termination.

Figure 1 illustrates the high-level structure of KaffeOS. User code executes in "user mode," as do some of the trusted runtime libraries and some of the garbage collection code. The remaining parts of the system (the rest of the runtime libraries and the garbage collector, as well as the virtual machine itself) must run in kernel mode to ensure their integrity. Note that "user mode" and "kernel mode" do not indicate a change in hardware privileges. Instead, they indicate different environments with respect to termination and resource consumption:

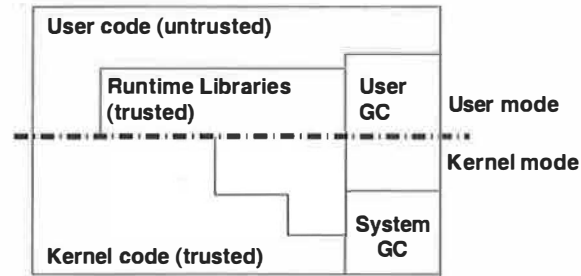


Figure 1: Structure of KaffeOS. System code is divided into kernel and user modes; user code all runs in user mode. In user mode, code can be terminated arbitrarily; in kernel mode, code cannot be terminated arbitrarily.

- Processes running in user mode can be terminated at will. Processes running in kernel mode cannot be terminated at an arbitrary time, because they must leave the kernel in a clean state.
- Resources consumed in user mode are always charged to a user process, and not to the system as a whole. Only in kernel mode can a process consume resources that are charged to the entire system, although typically such use is charged to the appropriate user process.

Such a structure echoes that of exokernels [18], where system-level code executes as a user-mode library. Note that a language-based system allows the kernel to trust user-mode code to a great extent, because type safety prevents user code from damaging any user-mode system code.

The KaffeOS kernel is structured so that it can handle termination requests and internal errors cleanly. Termination requests are deferred, so that a process cannot be terminated while manipulating kernel data structures. Kernel code must not abruptly terminate due to internal exceptions, for the same reason. Violations of these two restrictions are considered kernel bugs.

Others have suggested that depending on language-level exception handling is sufficient for safe termination. We disagree, because exceptions interact poorly with code in critical sections, which leaves shared data structures open to corruption. Even if termination requests were deferred during critical sections, one would need transactional support to ensure the integrity of mutually related data structures in the absence of a kernel. In addition, such an approach would be a confusing overloading of the concepts of mutual exclusion and deferred termination; preventing termination while any lock is held would also violate isolation.

Full reclamation of memory. Since Java is type-safe, it does not provide a primitive to reclaim memory. Instead, unreachable memory is freed by a garbage collector. We use the garbage collector to recover all the memory of a process when it terminates. Therefore, we must prevent situations where the collector cannot free a terminated process's objects because another process still holds references to them.

We use techniques from distributed garbage collection schemes [31] to restrict cross-process references. Distributed GC mechanisms are normally used to overcome the physical separation of machines and create the impression of a global shared heap. We use distributed GC mechanisms to manage multiple heaps in a single address space, so that they can be collected independently.

We use *write barriers* [43] to restrict writes. A write barrier is a check that happens on every pointer write to the heap. As we show in Section 4, the cost of using write barriers, although non-negligible, is reasonable.

Illegal cross-references are those that would prevent a process's memory from being reclaimed: for example, references from one user heap to another. Since those references cannot exist, it is possible to reclaim a process's heap as soon as the process is terminated. Writes that would create illegal cross-references are forbidden, and raise exceptions. We call such exceptions "segmentation violations." Although it may seem surprising that a type-safe language runtime could throw such a fault, it actually follows the analogy to traditional operating systems closely.

Unlike distributed garbage collection, in KaffeOS inter-heap cycles do not cause problems. The only form of inter-heap cycles that can occur are due to data structures that are split between a user heap and the kernel heap, since there can be no cycles that span multiple user heaps. Writes of user-heap references to kernel objects can only be done by trusted code. The kernel is coded so that it only writes a user-heap reference to a kernel object whose lifetime equals that of the user process: for example, the object that represents the process itself.

KaffeOS is intended to run on a wide range of systems. We assume that the platforms on which it runs will not necessarily have a hardware memory management unit under the control of KaffeOS. We also assume that the host may not have an operating system that supports virtual memory. For example, a Palm Pilot satisfies both of these assumptions. Under these assumptions, memory cannot simply be revoked by unmapping it.

Precise memory and CPU accounting. We account for memory and CPU on a per-process basis, so as to limit their consumption by buggy or possibly malicious

code. In addition, to prevent denial-of-service attacks, it is necessary to minimize the amount of time and memory spent servicing kernel requests.

Memory accounting is complete. It applies not only to objects at the Java level, but to all allocations done in the VM on behalf of a given process. In contrast, bytecode-rewriting approaches that do not modify the virtual machine, such as Jres [13, 14], can only account for object allocations.

We try to minimize the number of objects that are allocated on the kernel heap through careful coding of the kernel interfaces. For instance, consider a system call that creates a new process with a new heap: the process object itself, which is large, is allocated on the new heap. The handle that is returned to the creating process to control the new process is allocated on the creating process's heap. The kernel heap only maintains a small entry in a process table.

We increase the accuracy of CPU accounting by minimizing the time spent in non-preemptible sections of code. In addition, separately collecting user heaps and the kernel heap reduces the amount of time spent in the kernel. We again use write barriers: here, to detect cross-references from a user to the kernel heap, and vice versa. For each such reference, we create an *entry item* in the heap to which it points [31]. In addition, we create a special *exit item* in the original heap to remember the entry item created in the destination heap. Unlike distributed object systems such as Emerald [26], entry and exit items are not used for naming non-local objects; we only use them to decouple the garbage collection of different heaps.

Entry items are reference counted: they keep track of the number of exit items that point to them. The reference count of an entry item is decremented when an exit item is garbage collected. If an entry item's reference count reaches zero, the entry item is removed, and the referenced object can be garbage collected if it is not reachable through some other path.

A process's memory is reclaimed upon termination by merging its heap with the kernel heap. All exit items are destroyed at this point and the corresponding entry items are updated. The kernel heap's collector can then collect all of the memory, including memory on the kernel heap that was kept alive by the process. User-kernel cycles of garbage objects can be collected at this time. Note that a user process could attempt to create and kill and large number of new heaps to deny service to other processes. Such an attempt can only be prevented by imposing additional restrictions on the number or frequency with which a process may invoke kernel services.

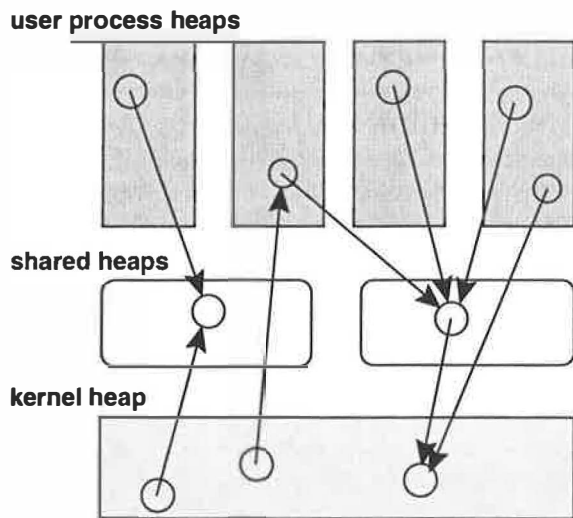


Figure 2: Heap structure in KaffeOS. The *kernel heap* can contain pointers into the user heaps, but the *shared heaps* and other user heaps cannot. User heaps can contain pointers into the kernel heap and shared heaps.

Direct sharing between processes. One of the reasons for using a language-based system is to allow for direct communication between applications. For example, the SPIN operating system allowed kernel extensions to communicate directly through pointers to memory. The design of KaffeOS retains this design principle. Figure 2 shows the different heaps in KaffeOS, and the kinds of inter-heap pointers that are legal.

In KaffeOS, a process can dynamically create a shared heap to communicate with other processes. A shared heap holds ordinary objects that can be accessed in the usual manner. Shared objects are not allowed to have pointers to objects on any user heap, because those pointers would prevent this user heap's full reclamation. This restriction is again enforced by write barriers; attempts to assign such pointers will result in an exception.

A shared heap has the following lifecycle. First, one process picks one or more shared types out of a central shared namespace, creates the heap, and loads the shared class or classes into it. While the heap is being created, the creator is charged for the whole heap. After the heap is populated with classes and objects, it is frozen and its size remains fixed for its lifetime. If other processes look up the shared heap, they are charged that amount. In this way, all sharers are charged for the heap. Processes exchange data by writing into and reading from the shared objects and by synchronizing on them in the usual way.

If a process drops all references to a shared heap, all exit items to that shared heap become unreachable. Af-

ter the process garbage collects the last exit item to a shared heap, that shared heap's memory is credited to the sharer's budget. When the last sharer drops all references to a shared heap, the shared heap becomes orphaned. The kernel garbage collector checks for orphaned shared heaps at the beginning of each GC cycle and merges them into the kernel heap.

This model guarantees three properties:

- All sharers are charged in full for a shared heap while they are holding onto the shared heap, whose size is fixed. As a result, sharers do not have to be charged asynchronously if another sharer exits. (If n sharers were each to pay only $1/n$ of the cost of a shared heap, when one sharer exited the others would have to be asynchronously charged $(1/n - 1) - (1/n)$ of the cost.)
- As already discussed, one process cannot use a shared object to keep objects in another process alive.
- Sharers are charged accurately for all metadata, such as internal class data structures. The metadata is also allocated on the shared heap. Unfortunately, this prevents us from applying any optimization that allocates data structures related to the shared heap lazily during execution.

Although process heaps can be scanned independently during GC, thread stacks still need to be scanned during GC for inter-heap references. Incremental schemes could be used to eliminate repeated scans of a stack [12], and a thread does not need to be scanned more than once while it is suspended. Some "GC crosstalk" between processes is still possible, because a process could create many threads in an effort to get the system to scan them all. We decided that the benefit of allowing direct sharing between processes is worth leaving open such a possibility.

Hierarchical memory management. We provide a simple hierarchical model for managing memory. Each heap is associated with a *memlimit*, which consists of an upper limit and a current use. Memlimits form a hierarchy: each one has a parent, except for a root memlimit. All memory allocated to the heap is debited from that memlimit, and memory collected from that heap is credited to the memlimit. This process of crediting/debiting is applied recursively to the node's parents.

A memlimit can be *hard* or *soft*. This attribute influences how credits and debits percolate up the hierarchy

of memlimits. A hard memlimit's maximum limit is immediately debited from its parent, which amounts to setting memory aside. Credits and debits are therefore not propagated past a hard limit. A soft memlimit's maximum limit, on the other hand, is just a limit—credits and debits of a soft memlimit's current usage are reflected in the parent.

Hard and soft limits allow different memory management strategies. Hard limits allow for memory reservations, but incur inefficient memory use if the limits are not used. Memory consumption matters, because we do not assume there is an underlying operating system; as a result, KaffeOS may manage physical memory. Soft limits allow the setting of a summary limit for multiple activities without incurring the inefficiencies of hard limits. They can be used to guard malicious or buggy applications where temporarily high memory usage can be tolerated.

Another application of soft limits is during the creation of shared heaps. Shared heaps are initially associated with a soft memlimit that is a child of the creating process heap's memlimit. In this way, they are separately accounted but still subject to their creator's memlimit, which ensures that they cannot grow to exceed their creator's ability to pay.

3 Discussion

The KaffeOS VM is built on top of the freely available Kaffe virtual machine, version 1.0b4 [42], which is roughly equivalent to JDK 1.1. In this section, we describe the specific issues that had to be dealt with in implementing KaffeOS. Many implementation decisions were driven by our desire to modify the Kaffe codebase as little as possible.

The primary purpose of KaffeOS is to run Java applications, which expect a well-defined environment of run-time services and libraries. We provide the standard Java API within KaffeOS.

We make use of various features of Java to support KaffeOS processes: Java class loaders, in particular, deserve some discussion. We also discuss our use of write barriers in more detail. Finally, we discuss some aspects of the Kaffe implementation that affect the performance that we can achieve with our KaffeOS prototype.

3.1 Write Barriers

An attempt to write a pointer to an object into a field of another object can have three different outcomes. In the common case, if a pointer to an object in the same heap is written, nothing needs to happen. If a pointer to a foreign heap is written, the write may either be aborted

and trigger an exception, or it will cause the creation of a pair of exit/entry items to keep track of that allowable inter-heap reference.

The option of aborting writes ensures that the separation that is necessary for full reclamation is maintained. A write barrier exception is either related to a foreign user heap, or to a shared heap. If a pointer to a foreign user heap is written, such a pointer must have been passed on the stack or in a register as a return value from a kernel call. Such write barrier violations indicate kernel bugs, since the kernel is not supposed to return foreign references to a user process. Write barrier violations on the shared heap, on the other hand, indicate attempts by user code to create a connection from the shared heap to the user heap. Such attempts may either be malicious, or a sign of a violation of the programming model imposed on shared objects.

Keeping track of entry and exit items ensures the separation that is necessary for independent garbage collection and the garbage collection of shared heaps. In the third case, the write barrier code will maintain entry and exit items. As a result, a local garbage collector will know to include all incoming references as roots in its garbage collection cycle.

Independent garbage collection, which relies on accurate bookkeeping of entry and exit items, is important in our model. Therefore, write barriers are necessary, if only to maintain entry and exit items. This statement holds true even if no shared heaps are being used.

Write barriers could only be optimized away if their outcome is known. Such is the case within a procedure if static analysis reveals that an assignment is between pointers on the same heap (for instance, if newly constructed objects are involved), or that a previous assignment must have had the same outcome. In addition, if a generational collector were used, it should be possible to reduce the write barrier penalty by combining the code for the generational and the inter-heap write barrier checks.

3.2 Namespaces

Separate namespaces are provided in Java through the use of class loaders [28]. A class loader is an object that acts as a name server for types, and maps names to types. We use the Java class loading mechanism to provide KaffeOS processes with different namespaces. This use of Java class loaders is not novel, but is important because we have tried to make use of existing Java mechanisms when possible. When we use standard Java mechanisms, we can easily ensure that we do not violate the language's semantics.

Processes may share types for two reasons: either be-

cause the class in question is part of the run-time library (i.e., is a system or kernel class), or because it is the type of a shared object located on a shared heap, which must be identical in the processes that have access to the shared heap. We refer to the former as system-shared, and the latter as user-shared. Process loaders delegate the loading of all shared types to a shared loader. If we did not delegate to a single loader, KaffeOS would need to support a much more complicated type system for its user-shared objects. Using one shared loader makes the namespace for user-shared classes global, which requires global and prior coordination between communicating partners. We use a simple naming convention for this shared namespace: the Java package `shared.*` contains all user-shared classes.

3.3 Java Class Libraries

To determine which classes can be system-shared, we examined each class in the Java standard libraries [10] to see how it interacted under the semantics of class loading. A class's members and their associated code are described by a sequence of bytes in a *class file*. Classes from identical class files that are loaded [28] by different class loaders are defined to be different in Java, even though they have identical behavior relative to the namespace defined by the loader that loaded them. We refer to such classes as *reloaded* classes. Reloaded classes are analogous to traditional shared libraries. Reloading a class gives each instance its own copies of static fields. In KaffeOS, Java classes could be reloaded; they could be modified to be shared across processes; or they could be used unchanged. For each class, we decided which alternative to choose, subject to two goals: to share as many classes as possible, but to make as few code changes as necessary.

Certain classes must be shared between processes. For example, the `java.lang.Object` class, which is the superclass of all object types, must be shared. If this type were not shared, it would not be possible for different processes to share generic objects! If a system-shared class uses static fields, and if these fields cannot be eliminated, they must be initialized with objects whose implementation is process-aware. Shared classes cannot directly refer to reloaded classes, because such references are represented using direct pointers by the run-time loader.

Non-shared classes should always be reloaded, so that each process gets its own instance. Reloaded classes do not share text in our current implementation, although they could. Because of some unfortunate decisions in the Java API design, some classes export static members as part of their public interface,

which forces those classes to be reloaded. For example, `java.io.FileDescriptor` must be reloaded, because it exports the public static variables `in`, `out`, and `err` (`stdin`, `stdout`, and `stderr`, respectively). Other, possibly more efficient, ways to accomplish the same thing as reloading exist [16], but their impact on type safety is not fully understood. Out of roughly 600 classes in the core Java libraries, we are able to safely system-share about 430 (72%) of them. The rest of the classes are reloaded.

3.4 Java Language Issues

A few language compatibility issues arose when building KaffeOS. For example, the Java language description assumes that all string literals are interned, and that equality can therefore be checked with a pointer comparison (the `==` operator). Unfortunately, to maintain such semantics, the interned string table would have to be a global (kernel) data structure—and user processes could allocate strings in an effort to make the kernel run out of memory. To deal with this problem, we chose to separately intern strings for each process. As a result, the Java language use of pointer comparison to check string equality does not work for strings that were created in different heaps, and the `equals` method must be used instead. It is impractical for the JVM to hide this semantic change from applications. However, this issue arises only in rare situations, and then only in KaffeOS-aware applications that directly use KaffeOS features.

3.5 Kaffe Limitations

Kaffe has relatively poor performance compared to commercial JVMs, for several reasons. First, its garbage collector is relatively primitive: it is a mark-and-sweep collector that is neither generational nor incremental. Second, it has a simple just-in-time bytecode compiler that translates each instruction individually. As a result, many unnecessary register spills and reloads are generated, and the native code that it produces is relatively poor.

4 Results

KaffeOS currently runs under Linux on the x86. We plan on porting it to the Itsy pocket computer from Compaq WRL; we have already ported Kaffe to the Itsy. To demonstrate the effectiveness of KaffeOS, we ran the following experiments:

- We measured three implementations of the write barrier. We ran the SPEC JVM98 benchmarks [35] on different configurations of KaffeOS, and the version of Kaffe on which it is based, and the IBM

JVM, which uses one of the fastest commercial JIT compilers [36] available. We must note that our results are not comparable with any published SPEC JVM98 metrics, as the measurements are not compliant with all of SPEC's run rules.

- We ran a servlet engine on KaffeOS to demonstrate that KaffeOS can prevent denial-of-service servlets from crashing a server. We also compared how the number of KaffeOS processes scales with how the number of OS processes scales.

Our measurements were all taken on a 800MHz "Katmai" Pentium III, with 256 Mbytes of SDRAM and a 133 MHz PCI bus, running Red Hat Linux 6.2. The processor has a split 32K L1 cache, and combined 256K L2 cache.

4.1 Write Barrier Implementations

To measure the cost of write barriers in KaffeOS, we implemented several versions:

- *No Write Barrier*. We execute without a write barrier, and run everything on the kernel heap.
- *No Heap Pointer*. At each heap pointer write, the write barrier consists of a call to a routine that finds an object's heap ID by looking at the page on which the object lies and performs the barrier checks. In order to avoid cache conflict misses, the actual heap ID is stored in a block descriptor that is not on the same page. This implementation takes 37 cycles with a hot cache.
- *Heap Pointer*. At each heap pointer write, the write barrier consists of a call to a routine that finds an object's heap ID in the object header and performs the barrier checks. This implementation takes only 11 cycles with a hot cache, but adds 4 bytes per object.
- *Fake Heap Pointer*. To measure the impact of the 4 bytes of padding in the *Heap Pointer* implementation, we use the third barrier implementation but add 4 bytes to each object.

The KaffeOS JIT compiler does not yet inline the write barrier routine. Inlining the write barrier would not necessarily improve performance, as it would lead to substantial code expansion.

We ran the SPEC JVM98 benchmark suite on IBM's JVM, on Kaffe00 and on KaffeOS with different implementations of the write barrier. Kaffe00 is the code base upon which the current version of KaffeOS is built. This version is from June 2000. We instrumented Kaffe00 and KaffeOS to estimate how many cycles are spent during

garbage collection. For IBM's JVM, we used a command line switch (-verbosegc) to obtain the number of milliseconds spent during garbage collection.

Figure 3 compares the results of our experiments. Each group of bars corresponds to IBM's JVM, Kaffe00, KaffeOS with no write barrier, KaffeOS with no heap pointer, KaffeOS with heap pointer, and KaffeOS with a fake heap pointer, in that order. The full bar displays the benchmark time as displayed by SPEC's JVM98 output. The upper part of the bar shows the time spent on those garbage collections that occurred during the actual benchmark run. Note that we excluded those collections that occurred while the SPEC test harness executed. The lower part of the bar represents the time not spent during garbage collection.

The time spent during garbage collection depends on the initial and maximum heap sizes, the allocation frequency, and the strategy used to decide when to collect. Kaffe00 and KaffeOS use a simple strategy: a collection is triggered whenever newly allocated memory exceeds 125% of the memory in use at the last GC. However, while Kaffe00 uses the memory occupied by objects as its measure, KaffeOS uses the number of pages as its measure, because KaffeOS's accounting mechanisms are designed to take internal fragmentation into account. In addition, KaffeOS decides when to collect for each heap separately. We do not know what strategy IBM's JVM uses, but its GC performance suggests that it is very aggressive at keeping its heap small.

Overall, IBM's JVM is between 2–5 times faster than Kaffe00; we will focus on the differences between Kaffe00 and the different versions of KaffeOS. While Kaffe00 and KaffeOS use different strategies for deciding when to collect, they use the same conservative non-moving collector. For this reason, we will focus on the time not spent on garbage collection.

The difference between Kaffe00 and KaffeOS *no write barrier* (excluding GC time) is minimal, which suggests that the changes done to Kaffe's run-time do not have significant performance impact. The difference between KaffeOS *no write barrier* and KaffeOS *no heap pointer* stems from the write barrier overhead, and is consistently below 7%.

Table 1 gives the number of write barriers that are executed in each of the SPEC benchmarks. When we compute the time to execute the write barriers by using the cycle counts for the barriers, we see that it is a fraction of the actual penalty. This discrepancy occurs because the microbenchmark uses a hot cache. For most benchmarks, the *heap pointer* optimization is effective in reducing the write barrier penalty to less than 5%. Excluding GC, KaffeOS *fake heap pointer* performs similarly

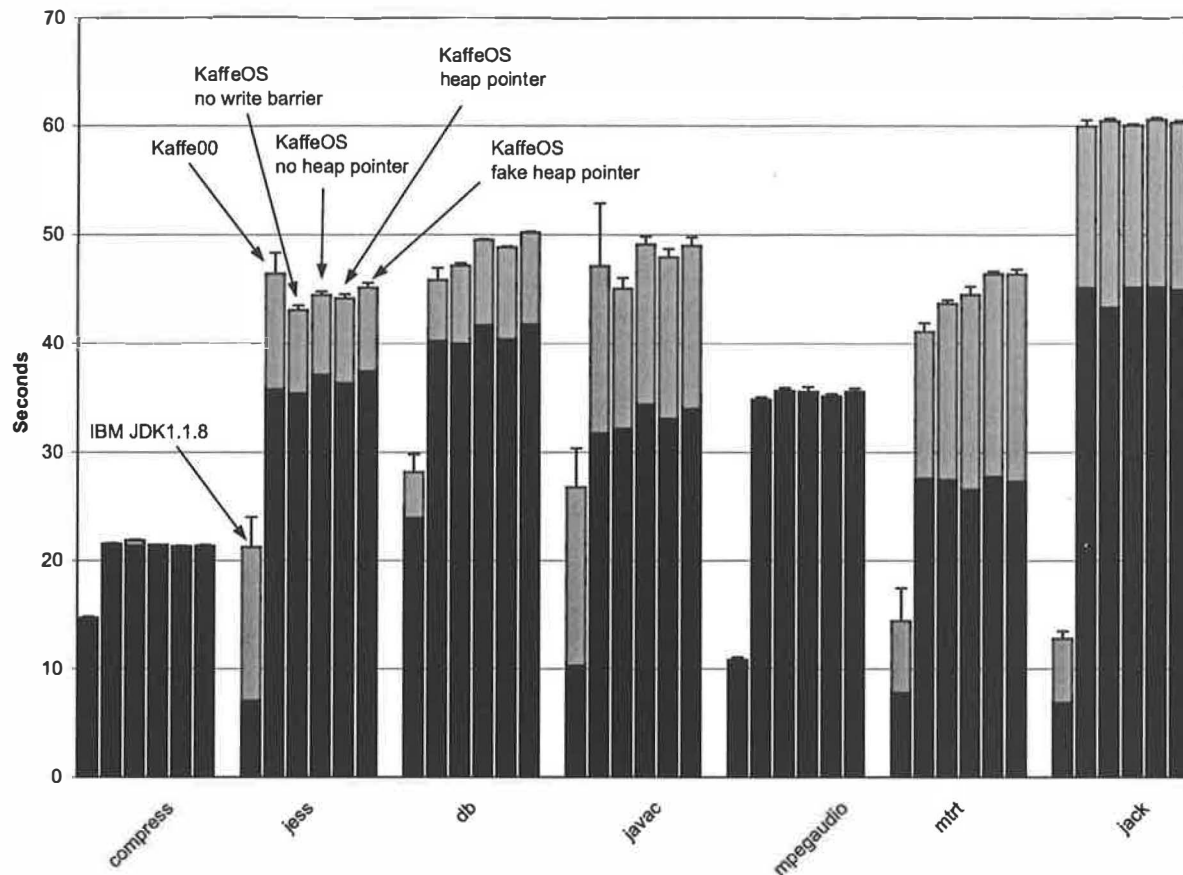


Figure 3: SPEC JVM98 run on various Java platforms. The error bars represent 95% confidence intervals. Each measurement is the result of three runs using SPEC's autorun mode. The upper part represents time spent in garbage collection.

Benchmark	Barriers	Time	Percent
compress	0.3M	0.014s	0.00%
jess	8.2M	0.38s	0.85%
db	30.4M	1.40s	2.84%
javac	21.1M	0.97s	1.97%
mpegaudio	5.8M	0.27s	0.75%
mrt	3.3M	0.15s	0.34%
jack	20.2M	0.93s	1.54%

Table 1: Number of write barriers executed for each SPEC JVM98 benchmark. "Time" is the total CPU cycle cost for the write barrier instructions, assuming the *No Heap Pointer* cost of 37 cycles; "percent" is the fraction of the *No Write Barrier* execution time.

to KaffeOS *no heap pointer*; however, its overall performance is lower because more time is spent during GC.

On a better system with a more effective JIT, the relative cost of using write barriers would increase. On the

other hand, a good JIT compiler could perform several kinds of optimizations to remove write barriers. A compiler should be able to remove redundant write barriers, along the lines of array bounds checking elimination. It could even perform method splitting to specialize methods, so as to remove useless barriers along frequently used call paths. We can only speculate as to what the performance penalty for implementing KaffeOS on the IBM JVM would be. Nevertheless, as we will show, the performance of KaffeOS is much better than that of the IBM JVM in the presence of uncooperative applications, despite the raw performance difference between them.

4.2 Servlet Engine

A Java servlet engine provides an environment for running Java programs (servlets) at a server. Their functionality subsumes that of CGI scripts at Web servers: for example, servlets may create dynamic content or run database queries. We use a **MemHog** servlet to measure

the effects of a denial-of-service attack. MemHog sits in a loop, repeatedly allocates memory, and keeps it from being garbage-collected.

We compared KaffeOS's ability to prevent the MemHog servlet from denying service with that of IBM's JVM. We used Apache 1.3.12, JServ 1.1 (Apache's servlet engine), and a free version of JSDK 2.0 to run our tests, without modification. JServ runs servlets in *servlet zones*, which are virtual servers. A single JServ instance can host one or more servlet zones. We ran each JServ in its own KaffeOS process. We compared KaffeOS against IBM's JVM, in two configurations: one servlet zone per JVM (IBM/1), and multiple servlet zones in one JVM (IBM/n). Due to time constraints, we used an earlier version of KaffeOS for these benchmarks. This version is about half as fast as the version used for the SPEC JVM benchmarks.

When simulating this denial-of-service attack, we did what a system administrator concerned with availability of his services would do: we restarted the JVM(s) and the KaffeOS process, respectively, whenever they crashed because of the effects caused by MemHog. In KaffeOS, MemHog will cause a single JServ to exit without affecting other JServs. If each JServ is started in its own IBM JVM, the whole JVM will eventually crash and be restarted. If all servlets are run in a single JServ on a single IBM JVM, the system runs out of memory in seemingly random places. This behavior resulted in exceptions that occurred at random places, which included the code that manipulated data structures that were shared between servlets in the surrounding JServ environment. Eventually, these data structures became corrupted, which results in an unhandled exception in JServ, or in some instances even a crash of the entire JVM.

Figure 4 illustrates the results of our experiments; note that the y axis uses a logarithmic scale. Running a separate KaffeOS process for each servlet has consistent performance, either with a MemHog running or without. This graph illustrates the most important feature of KaffeOS: that it can deliver consistent performance, even in the presence of uncooperative or malicious programs.

The graph shows that running each of the servlets in a single IBM JVM does not scale. This failure occurs because starting multiple JVMs eventually causes the machine to thrash. We estimate that each IBM JVM process takes about 2MB of virtual memory upon startup. We limited each JVM's heap size to 8MB in this configuration. An attempt to start 100 IBM JVMs rendered the machine inoperable.

If there are no uncooperative servlets running, using a single IBM JVM has the best performance. If there

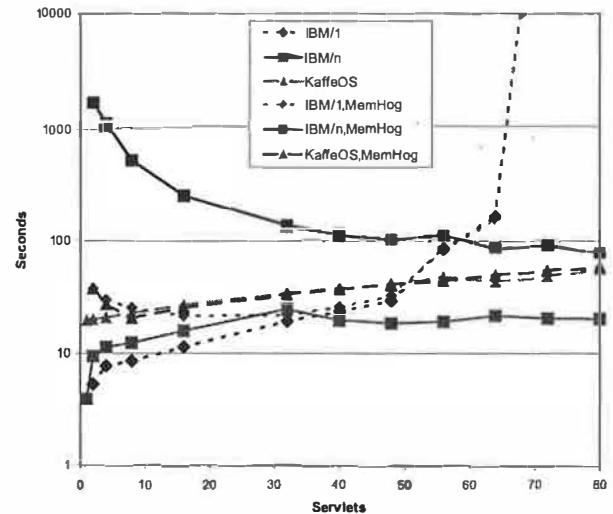


Figure 4: Scaling behavior of JVMs as the number of servlets increases. "IBM/1" means one IBM JVM per servlet; "IBM/n" means n servlets in one JVM. The "MemHog" measurements replace one of the good servlets with a MemHog. The y axis is the amount of time for the non-MemHog servlets to correctly respond to 1000 client requests.

is a MemHog servlet running, such a configuration has worse performance than KaffeOS—despite the fact that KaffeOS is several times slower for individual servlets! This degradation is caused by a lack of isolation between servlets. However, as the ratio of well-behaved servlets to malicious servlets increases, the scheduler will yield less often to the malicious servlet. Consequently, the service of IBM/n,MemHog improves as the number of servlets increases. This effect is an artifact of our experimental setup and cannot be reasonably used to defend against denial-of-service attacks.

Finally, we observe a slight service degradation as the number of KaffeOS processes increases. This degradation is likely due to inefficiencies in the user-mode threading system and scheduler.

5 Related Work

We classify the related work into three broad categories: extensible operating systems, resource management in operating systems, and Java extensions for resource management.

5.1 Extensible Operating Systems

Extensible operating systems have existed for many years. Most of them were not designed to protect against malicious users, although a number of them support

strong security features. None of them, however, provides strong resource controls. Pilot [32] and Cedar [38] were two of the earliest language-based systems. Their development at Xerox PARC predates a flurry of research in the 1990's on such systems. These systems include Oberon [44] and Juice [20], which are based on the Oberon language; SPIN [6], which is based on Modula-3; and Inferno [17], which is based on a language called Dis. Such systems can be viewed as single-address-space operating systems (see Opal [11]) that use type safety for protection.

VINO is a software-based (but not language-based) extensible system [34] that addresses resource management by wrapping kernel extensions within transactions. When an extension exceeds its resource limits, it can be safely aborted (even if it holds kernel locks) and its resources can be recovered. Transactions are a very effective mechanism, but they are also relatively heavyweight.

5.2 Resource Management

Several operating systems projects have focused on quality-of-service issues and real-time performance guarantees. Nemesis [27] is a single-address-space OS that focuses on quality-of-service for multimedia applications. Eclipse [8] introduced the concept of a *reservation domain*, which is a pool of guaranteed resources. Eclipse provides a guarantee of cumulative service, which means that processes execute at a predictable rate. It manages CPU, disk, and physical memory. Our work is orthogonal, because we examine the software mechanisms that are necessary to manage computational resources.

Recent work on resource management has examined different forms of abstractions for computational resources. Banga et al. [4] describe an abstraction called *resource containers*, which are effectively accounts from which resource usage can be debited. Resource containers are orthogonal to a process' protection domain: a process can contain multiple resource containers, and processes can share resource containers. In KaffeOS we have concentrated on the mechanisms to simply allow resource management; resource-container-like mechanisms could be added in the future.

5.3 Java Extensions

Besides KaffeOS, a number of other research systems have explored (or are currently exploring) the problem of supporting processes in Java.

The J-Kernel [23] and JRes [13, 14] projects at Cornell explore resource control issues without making changes to the Java virtual machine. The J-Kernel extends Java

by supporting capabilities between processes. These capabilities are indirection objects that can be used to isolate processes from each other. JRes extends the J-Kernel with a resource management interface whose implementation is portable across JVMs. The disadvantage of JRes (as compared to KaffeOS) is that JRes is a layer on top of a JVM; therefore, it cannot account for JVM resources consumed on the behalf of applications. Cornell is also exploring type systems that can support revocation directly [24].

Alta [39, 40] is a Java virtual machine that enforces resource controls based on a nested process model. The nested process model in Alta allows processes to control the resources and environment of other processes, including the class namespace. Additionally, Alta supports a more flexible sharing model that allows processes to directly share more than just objects of primitive types. Like KaffeOS, Alta is based on Kaffe, and, like KaffeOS, Alta provides support within the JVM for comprehensive memory accounting. However, Alta only provides a single, global garbage collector, so separation of garbage collection costs is not possible. For a more thorough discussion of Alta and the J-Kernel, see Back et al [1].

Balfanz and Gong [3] describe a multi-processing JVM developed to explore the security architecture ramifications of protecting applications from each other, as opposed to just protecting the system from applications. They identify several areas of the JDK that assume a single-application model, and propose extensions to the JDK to allow multiple applications and to provide inter-application security. The focus of their multi-processing JVM is to explore the applicability of the JDK security model to multi-processing, and they rely on the existing, limited JDK infrastructure for resource control.

Sun's original JavaOS [37] was a standalone OS written almost entirely in Java. It is described as a first-class OS for Java applications, but appears to provide a single JVM with little separation between applications. It was to be replaced by a new implementation termed "JavaOS for Business" that also ran only Java applications. "JavaOS for Consumers" is built on the Chorus microkernel OS [33] to achieve real-time properties needed in embedded systems. Both of these systems apparently require a separate JVM for each Java application, and all run in supervisor mode.

Joust [22], a JVM integrated into the Scout operating system [30], provides control over CPU time and network bandwidth. To do so, it uses Scout's path abstraction. However, Joust does not support memory limits on applications.

The Open Group's Conversant system [5] is another project that modifies a JVM to provide processes. It pro-

vides each process with a separate address range (within a single Mach task), a separate heap, and a separate garbage collection thread. Conversant does not support sharing between processes, unlike KaffeOS, Alta, and the J-Kernel.

The Real-Time for Java Experts Group [7] has published a proposal to add real-time extensions to Java. This proposal provides for *scoped memory* areas with a limited lifetime, which can be implemented using multiple heaps that resemble KaffeOS's heaps. The proposal also dictates the use of write barriers to prevent pointer assignments to objects in short-lived inner scopes. Real-Time Java's main focus is to ensure predictable garbage collection characteristics in order to meet real-time guarantees; it does not address untrusted applications.

6 Conclusions

We have described the design and implementation of KaffeOS, a Java virtual machine that supports the operating system abstraction of *process*. KaffeOS enables processes to be isolated from each other, to have their resources controlled, and still share objects directly. Processes enable the following important features:

- The resource demands of Java processes can be accounted for separately, including memory consumption and GC time.
- Java processes can be terminated if their resource demands are too high, without damaging the system.
- Termination reclaims the resources of the terminated Java process.

These features enable KaffeOS to run untrusted code safely, because it can prevent simple denial-of-service attacks that would disable standard JVMs. The cost of these features, relative to Kaffe, is reasonable. Because Kaffe's performance is poor compared to commercial JVMs, it is difficult to estimate the cost of adding such features to a commercial JVM—but we believe that the overhead should not be excessive. Finally, even though KaffeOS is substantially slower than commercial JVMs, it exhibits much better performance scaling in the presence of uncooperative code.

Acknowledgements

We thank many members of the Flux group, especially Patrick Tullmann, for enlightening discussion, comments on earlier drafts, and assistance in running some of the experiments. We would like to thank Tim Stack for his help in upgrading the KaffeOS code base to a

more recent version of Kaffe, and Jason Baker for some earlier implementation work. We are also grateful to those anonymous referees whose questions and comments helped us improve the paper, and to our shepherd David Culler.

References

- [1] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Techniques for the Design of Java Operating Systems. In *Proc. of the USENIX 2000 Annual Technical Conf.*, pages 197–210, San Diego, CA, June 2000. USENIX Association.
- [2] G. V. Back and W. C. Hsieh. Drawing the red line in Java. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, Mar. 1999. IEEE Computer Society.
- [3] D. Balfanz and L. Gong. Experience with Secure Multi-Processing in Java. In *Proc. of the Eighteenth International Conf. on Distributed Computing Systems*, May 1998.
- [4] G. Banga, P. Druschel, and J. C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation*, pages 45–58, New Orleans, LA, Feb. 1999.
- [5] P. Bernadat, L. Feeney, D. Lambright, and F. Travostino. Java Sandboxes meet Service Guarantees: Secure Partitioning of CPU and Memory. Technical Report TOGRI-TR9805, The Open Group Research Institute, June 1998.
- [6] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Ficuzynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, CO, Dec. 3–6, 1995.
- [7] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.
- [8] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. The Eclipse Operating System: Providing Quality of Service via Reservation Domains. In *Proc. of the 1998 USENIX Annual Technical Conf.*, pages 235–246, New Orleans, LA, June 1998.

- [9] P. Chan and R. Lee. *The Java Class Libraries: Volume 2. The Java Series*. Addison-Wesley, second edition, November 1998.
- [10] P. Chan, R. Lee, and D. Kramer. *The Java Class Libraries: Volume 1. The Java Series*. Addison-Wesley, second edition, November 1998.
- [11] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and Protection in a Single-Address-Space Operating System. *ACM Transactions on Computer Systems*, 12(4):271–307, 1994.
- [12] P. Cheng, R. Harper, and P. Lee. Generational Stack Collection and Profile-Driven Pretenuring. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 162–173, Montreal, Canada, June 1998.
- [13] G. Czajkowski, C.-C. Chang, C. Hawblitzel, D. Hu, and T. von Eicken. Resource Management for Extensible Internet Servers. In *Proceedings of the 8th ACM SIGOPS European Workshop*, Sintra, Portugal, Sept. 1998.
- [14] G. Czajkowski and T. von Eicken. JRes: A Resource Accounting Interface for Java. In *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, Vancouver, Canada, Oct. 1998. ACM.
- [15] Delivering the Promise of Internet Computing: Integrating Java With Oracle8i. http://www.oracle.com/database/documents/-delivering_the_promise_twp.pdf, Apr. 1999.
- [16] D. Dillenberger, R. Bordawekar, C. W. Clark, D. Durand, D. Emmes, O. Gohda, S. Howard, M. F. Oliver, F. Samuel, and R. W. S. John. Building a Java virtual machine for server applications: The JVM on OS/390. *IBM Systems Journal*, 39(1):194–210, 2000. Reprint Order No. G321-5723.
- [17] S. Dorward, R. Pike, D. L. Presotto, D. Ritchie, H. Trickey, and P. Winterbottom. Inferno. In *Proceedings of the 42nd IEEE Computer Society International Conference*, San Jose, CA, February 1997.
- [18] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Dec. 1995.
- [19] K. Farkas, J. Flinn, G. Back, D. Grunwald, and J. Anderson. Quantifying the Energy Consumption of a Pocket Computer and a Java Virtual Machine. In *Proceedings of SIGMETRICS '00*, page to appear, Santa Clara, CA, June 2000.
- [20] M. Franz. Beyond Java: An Infrastructure for High-Performance Mobile Code on the World Wide Web. In S. Lobodzinski and I. Tomek, editors, *Proceedings of WebNet '97*, pages 33–38, October 1997.
- [21] L. Gorrie. Echidna — A free multiprocess system in Java. <http://www.javagroup.org/echidna/>.
- [22] J. H. Hartman et al. Joust: A Platform for Communication-Oriented Liquid Software. Technical Report 97-16, Univ. of Arizona, CS Dept., Dec. 1997.
- [23] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing Multiple Protection Domains in Java. In *Proc. of the USENIX 1998 Annual Technical Conf.*, pages 259–270, New Orleans, LA, June 1998.
- [24] C. Hawblitzel and T. von Eicken. Type System Support for Dynamic Revocation. In *Second Workshop on Compiler Support for System Software*, Atlanta, GA, May 1999.
- [25] T. Jaeger, J. Liedtke, and N. Islam. Operating System Protection for Fine-Grained Programs. In *Proc. of the Seventh USENIX Security Symposium*, pages 143–157, Jan. 1998.
- [26] E. Jul, H. Levy, N. Hutchison, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.
- [27] I. M. Leslie, D. McAuley, R. J. Black, T. Roscoe, P. R. Barham, D. M. Evers, R. Fairbairns, and E. A. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, Sept. 1996.
- [28] S. Liang and G. Bracha. Dynamic Class Loading in the Java Virtual Machine. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications '98*, pages 36–44, Vancouver, Canada, Oct. 1998.
- [29] D. Malkhi, M. K. Reiter, and A. D. Rubin. Secure Execution of Java Applets using a Remote Playground. In *Proc. of the 1998 IEEE Symposium on Security and Privacy*, pages 40–51, Oakland, CA, May 1998.

- [30] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, pages 153–167, Seattle, WA, Oct. 1996. USENIX Association.
- [31] D. Plainfossé and M. Shapiro. A Survey of Distributed Garbage Collection Techniques. In *Proceedings of the 1995 International Workshop on Memory Management*, Kinross, Scotland, Sept. 1995.
- [32] D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, and S. C. Purcell. Pilot: An Operating System for a Personal Computer. *Communications of the ACM*, 23(2):81–92, 1980.
- [33] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. The Chorus Distributed Operating System. *Computing Systems*, 1(4):287–338, Dec. 1989.
- [34] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, WA, Oct. 1996. USENIX Association.
- [35] SPEC JVM98 Benchmarks. <http://www.spec.org/osg/jvm98/>.
- [36] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000.
- [37] Sun Microsystems, Inc. JavaOS: A Stand-alone Java Environment, Feb. 1997. <http://www.javasoft.com/products/javaos/javaos-white.html>.
- [38] D. C. Swinehart, P. T. Zellweger, R. J. Beach, and R. B. Hagmann. A Structural View of the Cedar Programming Environment. *ACM Transactions on Programming Languages and Systems*, 8(4):419–490, Oct. 1986.
- [39] P. Tullmann and J. Lepreau. Nested Java Processes: OS Structure for Mobile Code. In *Proc. of the Eighth ACM SIGOPS European Workshop*, pages 111–117, Sintra, Portugal, Sept. 1998.
- [40] P. A. Tullmann. The Alta Operating System. Master's thesis, University of Utah, 1999. 104 pages. Also available at <http://www.cs.utah.edu/flux/papers/tullmann-thesis-abs.html>.
- [41] D. J. Wetherall, J. Guttag, and D. L. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *Proceedings of IEEE OPENARCH '98*, San Francisco, CA, April 1998.
- [42] T. Wilkinson. Kaffe – a Java virtual machine. <http://www.transvirtual.com>.
- [43] P. R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proceedings of the 1992 International Workshop on Memory Management*, St. Malo, France, Sept. 1992. Expanded version, submitted to Computing Surveys.
- [44] N. Wirth and J. Gutknecht. *Project Oberon*. ACM Press, New York, NY, 1992.

Knit: Component Composition for Systems Software

Alastair Reid

Matthew Flatt

Leigh Stoller

Jay Lepreau

Eric Eide

University of Utah, School of Computing

{reid,mflatt,stoller,lepreau,eeide}@cs.utah.edu <http://www.cs.utah.edu/flux/>

Abstract

Knit is a new component definition and linking language for systems code. *Knit* helps make C code more understandable and reusable by third parties, helps eliminate much of the performance overhead of componentization, detects subtle errors in component composition that cannot be caught with normal component type systems, and provides a foundation for developing future analyses over C-based components, such as cross-component optimization. The language is especially designed for use with component kits, where standard linking tools provide inadequate support for component configuration. In particular, we developed *Knit* for use with the OSKit, a large collection of components for building low-level systems. However, *Knit* is not OSKit-specific, and we have implemented parts of the Click modular router in terms of *Knit* components to illustrate the expressiveness and flexibility of our language. This paper provides an overview of the *Knit* language and its applications.

1 Components for Systems Software

Software components can reduce development time by providing programmers with prepackaged chunks of reusable code. The key to making software components work is to define components that are general enough to be useful in many contexts, but simple enough that programmers can understand and use the components' interfaces.

Historically, developers have seen great success with components only in the limited form of libraries. The implementor of a library provides services to unknown clients, but builds on top of an existing, *known* layer of services. For example, the X11 library builds on the C library. A component implementor, in contrast, provides services to an unknown client while simulta-

neously importing services from an *unknown* supplier. Such components are more flexible than libraries, because they are more highly parameterized, but they are also more difficult to implement and link together. Despite the difficulty of implementing general components, an ever-growing pressure to reuse code drives the development of general component collections such as the OSKit [10].

Existing compilers and linkers for systems software provide poor support for components because these tools are designed for library-based software. For example, to reference external interfaces, a client source file must refer to a specific implementation's header files, instead of declaring only the services it needs. Compiled objects refer to imports within a global space of names, implicitly requiring all clients that need a definition for some name to receive the same implementation of that name. Also, to mitigate the performance penalty of abstraction, library header files often include specific function implementations to be inlined into client code. All of these factors tend to tie client code to specific library implementations, rather than allowing the client to remain abstract with respect to the services it requires.

A programmer can fight the system, and—by careful use of `#include` redirection, preprocessor magic, and name mangling in object files—manage to keep code abstracted from its suppliers. Standard programming tools offer the programmer little help, however, and the burden of ensuring that components are properly linked is again left to the programmer. This is unfortunate, considering that component interfaces are inherently more complex than library interfaces. Indeed, attempting to use such techniques while developing the OSKit has been a persistent source of problems, both for ourselves as developers and for OSKit users.

We have developed a new module language and toolset for managing systems components called *Knit*. *Knit* is based on *units* [8, 9], a model of components in the spirit of the Mesa [23] and Modula-3 [14] module languages. In addition to bringing state-of-the-art module technology to C programs, *Knit* provides features of particular use in the design and implementation of complex, low-level systems:

This research was largely supported by the Defense Advanced Research Projects Agency and the Air Force Research Laboratory, under agreement numbers F30602-99-1-0503 and F33615-00-C-1696. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon.

US Mail contact address: School of Computing, 50 S. Central Campus Drive, Room 3190, University of Utah, SLC, UT 84112-9205.

- Knit provides automatic scheduling of component initialization and finalization, even in the presence of mutual dependencies among components. This scheduling is possible because each component describes, in addition to its import requirements, specific initialization requirements.
- Knit's constraint-checking system allows programmers to define domain-specific architectural invariants and to check that systems built with Knit satisfy those invariants. For example, we have used Knit to check that code executing without a process context will never call code that requires a process context.
- Knit can inline functions across component boundaries, thus reducing one of the basic performance overheads of componentization and encouraging smaller and more reusable components.

We have specifically designed Knit so that linking specifications are static, and so that Knit tools can operate on components in source form as well as compiled form. Although dynamic linking and separate compilation fit naturally within our core component model, our immediate interests lie elsewhere. We are concerned with low-level systems software that is inherently static and amenable to global analysis after it is configured, but where flexibility and assurance are crucial during the configuration stage.

Knit's primary target application is the OSKit, a collection of components for building custom operating systems and extending existing systems. Knit is not OSKit-specific, however. As an additional example for Knit, we implemented part of MIT's Click modular router [25] in terms of Knit components, showing how Knit can help express both Click's component implementations and its linking language.

In the following sections we explain the problems with existing linking tools (Section 2) and present our improved language (Section 3), including its constraint system for detecting component mismatches (Section 4). We describe our initial experience with Knit in the OSKit and a subset of Click (Section 5). We then describe our preliminary work on reducing the performance overhead of componentization (Section 6). Finally, we describe related work (Section 7).

2 Linking Components

With existing technology, the two main options for structuring component-based, low-level systems are to implement components as object files linked by `ld` (the standard Unix linker), or to implement components as objects in an object-oriented language (or, equivalently, COM objects). Neither is satisfactory from the point of

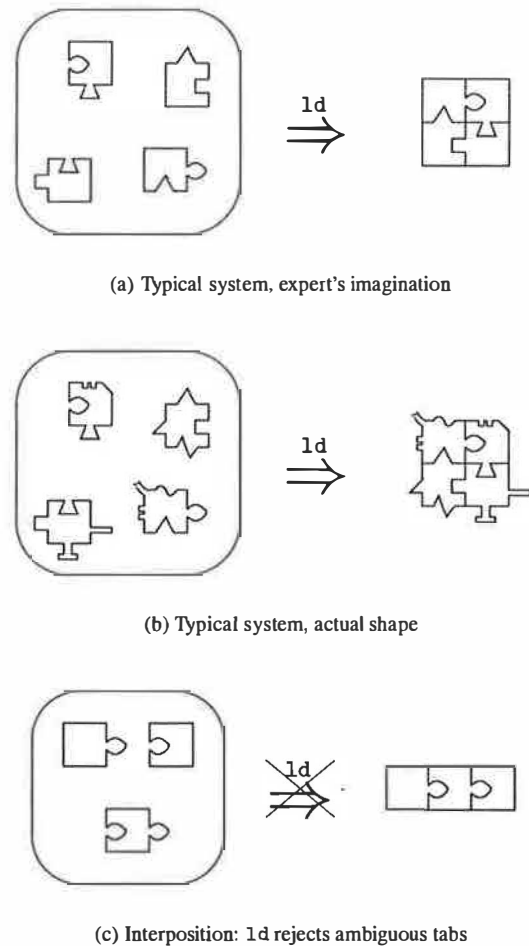


Figure 1: Linking with `ld`

view of component kits; the reasons given in Section 2.1 and Section 2.2 reflect our experience in trying each within the OSKit. Though our experience with standard linking may be unsurprising, our analysis helps to illuminate the parts of our Knit linking model, developed specifically for component programming, described in Section 2.3.

2.1 Conventional Linking

Figure 1(a) illustrates the way that a typical program is linked, through the eyes of an expert who understands the entire program. Each puzzle piece in the figure represents an object (`.o`) file. A tab on a puzzle piece is a global variable or function provided by the object. A notch in a puzzle piece is a global variable or function used by the object that must be defined elsewhere in the program. Differently shaped tabs and notches indicate differently named variables and functions.

The balloon on the left-hand side represents the col-

lection of object files that are linked to create the program. The programmer provides this collection of files to the linker as a grab bag of objects. The right-hand side of the figure shows the linker's output. The linker matches all of the tabs with notches, fitting the whole puzzle together in the obvious way.

This bag-of-objects approach to linking is flexible. A programmer can reuse any puzzle piece in any other program, as long as the piece's tabs and notches match other pieces' notches and tabs. Similarly, any piece of the original puzzle can be replaced by a different piece as long as it has the same tabs and notches. Linkers support various protocols for "overriding" a puzzle piece in certain bags (i.e., archive files) without having to modify the bag itself. In all cases, the tedious task of matching tabs and notches is automated by the linker.

The bag-of-objects approach to linking has a number of practical drawbacks, however. Figure 1(b) illustrates the same linking process as Figure 1(a), but this time more realistically, through the eyes of a programmer who is new to the program. Whereas the expert imagines the program to be composed of well-defined pieces that fit together in an obvious way, the actual code contains many irrelevant tabs and notches (e.g., a "global" variable that is actually local to that component, or a spurious and unused extern declaration) that obscure a piece's role in the overall program. Indeed, some edges are neither clearly tabs nor clearly notches (e.g., an uninitialized global variable might implement an export or an import). If the new programmer wishes to replace a piece of the puzzle, it may not be clear which tabs and notches of the old piece must be imitated by the new piece, and which tabs and notches of the old piece were mere implementation artifacts.

The bag-of-objects approach also has a significant technical limitation when creating new programs from existing pieces. Figure 1(c) illustrates the limitation: a new component is to be interposed between two existing components, perhaps to log all calls between the top-right component and the top-left component. To achieve this interposition, the tabs and notches of the bottom piece have the same shapes as the tabs and notches of the top pieces. Now, however, the bag of objects does not provide enough linking information to allow ld to resolve the ambiguous tabs and notches. (Should the linker build a two-piece or a three-piece puzzle?) The programmer will be forced to modify at least two of the components (perhaps with preprocessor tricks) to change the shape of some tabs and notches.

2.2 Object-Based Linking

At the opposite end of the spectrum from ld, components can be implemented as objects in an object-oriented language or framework. In this approach, the

links among components are defined by arbitrary code that passes object references around. This is the view of components implemented by object-based frameworks such as COM [22] and CORBA [26], and by some languages such as Limbo [6], which relies heavily on dynamic (module-oriented) linking.

Although linking via arbitrary run-time code is especially flexible, it is too dynamic for most uses of components in systems software. Fundamentally, object-oriented constructs are ill-suited for organizing code at the module level [7, 30]. Although classes and objects elegantly express run-time concepts, such as files and network connections, they do not provide the structure needed by programmers (and analysis tools) to organize and understand the static architecture of a program.

Symptoms of misusing objects as components include the late discovery of errors, difficulty in tracing the source of link errors, a performance overhead due to virtual function calls, and a high programmer overhead in terms of manipulating reference counts. Code for linking components is intermingled with regular program statements, making the code difficult for both humans and machines to analyze. Even typechecking is of limited use, since object-based code uses many dynamic typechecks (i.e., downcasts) to verify that components have the expected types, and must be prepared to recover if this is not so. These problems all stem from using a dynamic mechanism (objects) to build systems in which the connections between components change rarely, if ever, after the system is configured and initialized.

In short, object-based component languages offer little help to the programmer in ensuring that components are linked together properly. While objects can serve a useful and important role in implementing data structures, they do as much harm as good at the component level.

2.3 Unit Linking

The linking model for *units* [8,9] eschews the bag of objects in favor of explicit, programmer-directed linking. It also avoids the excessive dynamism and intractable analysis of object-based linking by keeping the linking specification separate from (and simpler than) the core programming language. The model builds on pioneering research for component-friendly modules in Mesa [23], functors in ML [21], and generic packages in Modula-3 [14] and Ada95 [18].

Linking with units includes specific linking instructions that connect each notch to its matching tab. The linking specification may be hierarchical, in that a subset of the objects can be linked to form a larger object (or puzzle piece), which is then available for further linking.

Unit linking can thus express the use pattern in Figure 1(c) that is impossible with ld. Furthermore,

unlike object-based linking, a program's explicit linking specification helps programmers understand the interface of each component and the role of each component in the overall program. The program's linking hierarchy serves as a roadmap to guide a new programmer through the program structure.

Unit linking also extends more naturally to cross-component optimization than do ld or object-based linking. The interfaces and linking graph for a program can be specified in advance, before any of the individual components are compiled. The compiler can then combine the linking graph with the source code for surrounding components to specialize the compilation of an individual component. The linking hierarchy may also provide a natural partitioning of components into groups to be compiled with cross-component optimization, thus limiting the need to know the *entire* program to perform optimizations.

The static nature of unit linking specifications makes them amenable to various forms of analysis, such as ensuring that components are linked in a way that satisfies certain type and specification constraints. For example, details on the adaptation of expressive type languages (such as that of ML) to units can be found in Flatt and Felleisen's original units paper [9]. This support for static analysis provides a foundation for applying current and future research to systems components.

3 Units for C

In this section we describe our unit model in more detail, especially as it applies to C code. We first look at a simplified model that covers component imports, exports, and linking. We then refine the model to address the complications of real code, including initialization constraints.

3.1 Simplified Model

Our linking model consists of two kinds of units: *atomic units*, which are like the smallest puzzle pieces, and *compound units*, which are like puzzle pieces that contain other puzzle pieces. Figure 2 expands the model of a unit given in Section 2.3 to a more concrete representation for a unit implemented in C.¹ According to this representation, every atomic unit has three parts:

1. A set of *imports* (the top part of the box), which are the names of functions and variables that will be supplied to the unit by another unit.
2. A set of *exports* (the bottom part of the box), which are the names of functions and variables that are defined by the unit and provided for use by other units.

¹Knit actually relies on a textual language for unit descriptions, as shown in Section 3.3.

3. A set of top-level C declarations (the middle part of the box), which must include a definition for each exported name, and may include uses of each imported name. Defined names that are not exported will be hidden from all other units.

The example unit in Figure 2 shows a component within a Web server, as it might be implemented with the OSKit. The component exports a `serve_web` function that inspects a given URL and dispatches to either `serve_file` or `serve_cgi`, depending on whether the URL refers to a file or CGI script.

Atomic units are linked together to form compound units, as illustrated in Figure 3. A compound unit has a set of imports (the top part of the outer box) that can be propagated to the imports of units linked to form the compound unit. The compound unit explicitly specifies how imports are propagated to other units; these propagations can be visualized as arrows. A compound unit also has a set of exports (the bottom part of the outer box) that are drawn from the exports of the units linked to form the compound unit. The compound unit explicitly specifies which exports are to be propagated. Because all connections are *explicitly* specified, arrows can connect imports and exports with different names, allowing each unit to use locally meaningful names without the danger of clashes in a global namespace.

The imports of the linked units that are not satisfied by imports of the compound unit must be satisfied by linking them to the exports of other units within the compound unit. As before, the compound unit defines these links. The units linked together in a compound unit need not be atomic units; they can be compound units as well.

The example in Figure 3 links the previous example unit with another unit that logs requested URLs. The original `serve_web` function is wrapped with a new one, `serve_logged`, to perform the logging. The resulting compound unit still requires `serve_file` and `serve_cgi` to be provided by other units, and also requires functions for manipulating files. The compound unit's export is the logged version of `serve_web`.

3.2 Realistic Model

To make units practical for real systems code, we must enhance the simple unit model in a number of ways. Figure 4 shows a more realistic model of units in Knit.

First, instead of importing and exporting individual function names, Knit units import and export names in *bundles*. For example, the `stdio` bundle groups `fopen`, `fprintf`, and many other functions. Grouping names into bundles makes unit definitions more concise and lets programmers define components in terms of standardized bundles.

Second, the simplified model shows source code inlined in the unit's definition, but it is more practical to

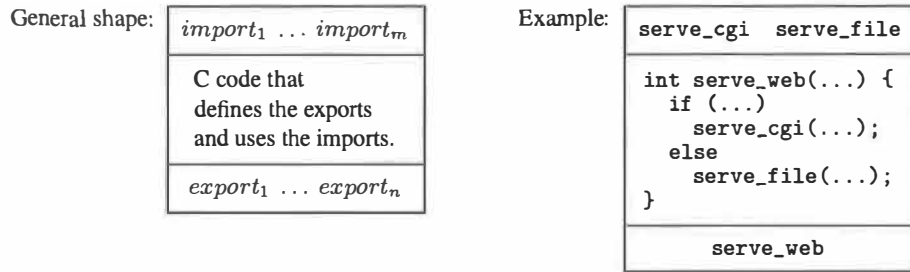


Figure 2: A unit implemented in C, ideally

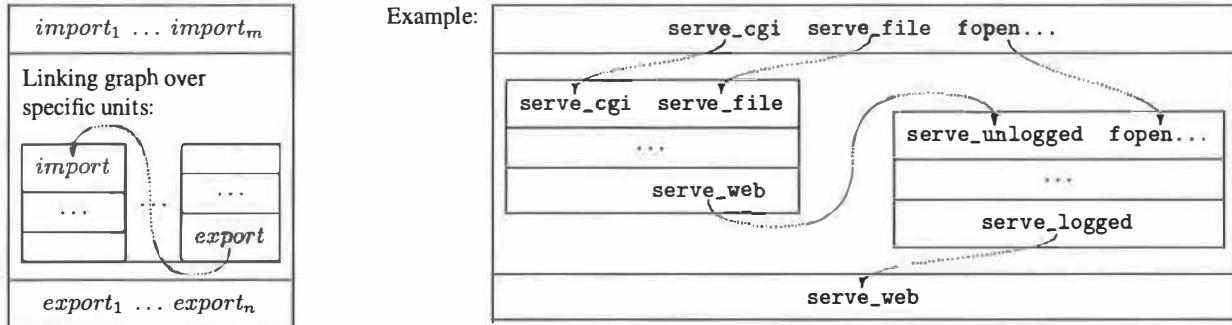


Figure 3: A compound unit, ideally

define units by referring to one or more external C files.² To convert the source code to compiled object files, Knit needs both the source files and their compilation flags. Figure 4 shows how the logging component's content is created by compiling `log.c` using the include directory `oskit/include`.

Third, realistic systems components have complex initialization dependencies. If there were no cyclic import relations among components, then initializations could be scheduled according to the import graph. In practice, however, cyclic imports are common, so the programmer must occasionally provide fine-grained dependency information to break cycles. A Knit unit therefore provides an explicit declaration of the unit's initialization functions, plus information about the dependencies of exports and initializers on imports. Based on these declarations, Knit automatically schedules calls to component initializers. Finalizers are treated analogously to initializers, but are called after the corresponding exports are no longer needed.

For example, the logging unit in Figure 4 defines an `open_log` function to initialize the component and a `close_log` function to finalize it. The functions ex-

ported in the `serveLog` bundle are declared to call the functions in the imported `serveWeb` and `stdio` bundles, and the initialization and finalization functions `open_log` and `close_log` rely only on the functions in the `stdio` bundle.

The `open_log` and `close_log` dependency declarations reveal a subtlety in declaring initialization constraints. The declaration "`serveLog needs stdio`" indicates that `stdio` must be initialized before any function in the bundle `serveLog` is called. However, this declaration alone does not constrain the order of initialization between the logging component and the standard I/O component; it simply says that both must be initialized before a `serveLog` function is used. In contrast, the declaration "`open_log needs stdio`" ensures that the standard I/O component is initialized before the logging component, because the logging component's initialization relies on standard I/O functions. The distinction between dependency levels is crucial to avoid over-constraining the initialization order.

A final feature needed by real units is that imports and exports may need to be renamed in order to associate Knit symbols with the identifiers used in the actual (C) implementation of a unit. For example, a serial console implementation might define a function `serial_putchar`, but export it as `putchar` to match a

²Knit can actually work with C, assembly, and object code. Extending Knit to handle C++, or any other language that compiles to `.o` with C-like conventions, would be straightforward but time-consuming.

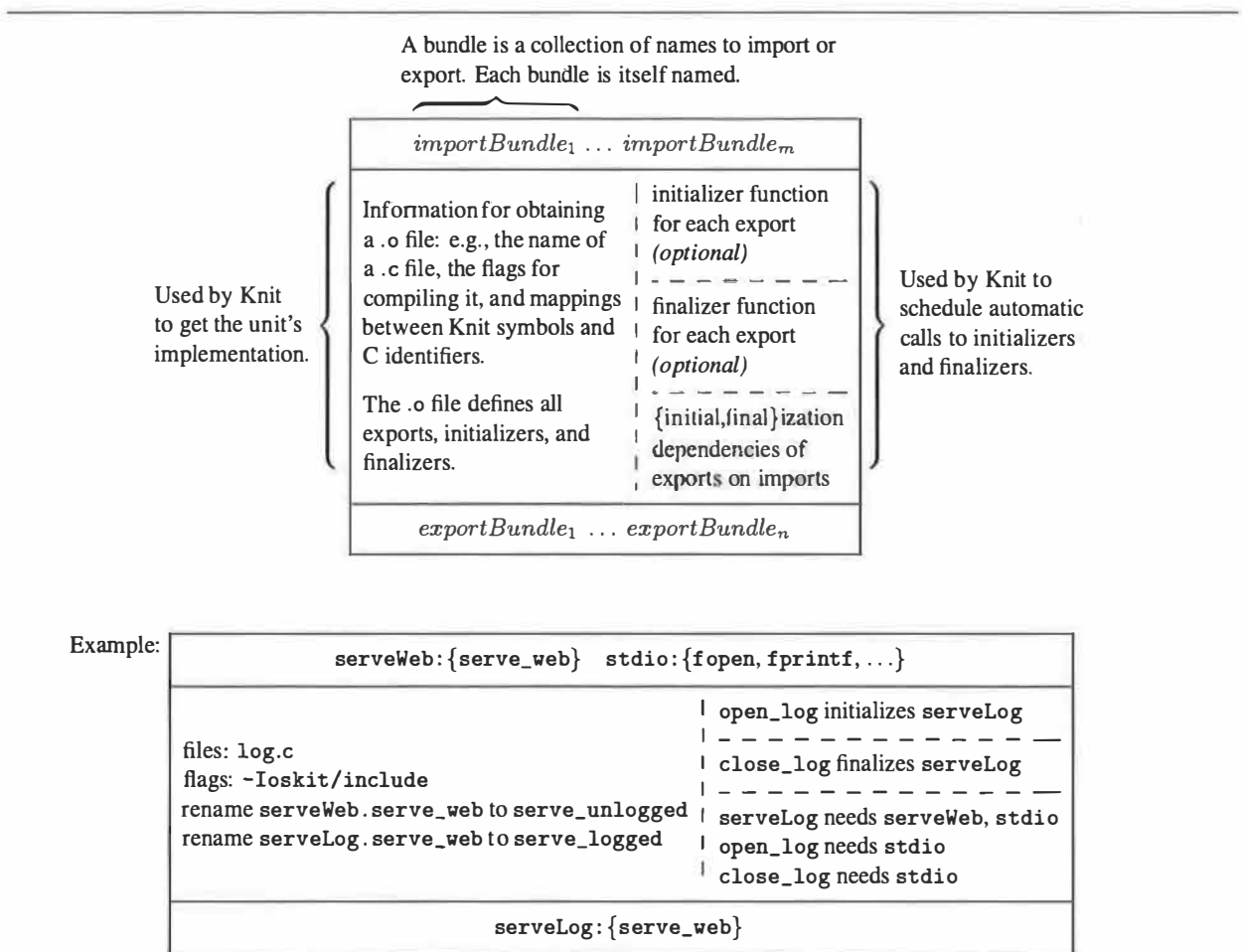


Figure 4: A unit implemented in C, more realistically. It exports a bundle `serveLog` containing the single function `serve_web`.

more generic unit interface. Another example would be a unit that both imports and exports a particular bundle type, a pattern that occurs frequently in units designed to “wrap” or interpose on other units. The logging unit shown in Figure 4 is such a unit; the implementation—the code in `log.c`—must be able to distinguish between the imported `serve_web` and the exported `serve_web` functions. This distinction is made by renaming the import or the export (or both) so that the functions have different names in the C code.

3.3 Example Code

Due to space constraints, we omit a full description of the Knit syntax. Nevertheless, to give a sense of Knit’s current concrete syntax, we show how to express the running example.³ For the sake of exposition and maintaining correspondence with the pictures, we have avoided

³The syntax continues to evolve as we gain experience. Also, although we do not currently have a graphical tool for Knit, we are considering implementing one in the future.

some syntactic sugar that can shorten real unit definitions.

Figure 5 shows Knit declarations for the Web server and logging components, plus a compound unit linking them together. Before defining the units, the code defines bundle types `Serve` and `Stdio` (artificially brief in this example) and a set of compiler flags, `CFlags`. These declarations are used within the unit definitions.

As in the graphical notation in Figure 4, the Web unit in Figure 5 imports two functions, one for serving files and another for serving CGI scripts. The notation for both imports and exports declares a local name within the unit (the left hand side of the colon) and specifies the type of the bundle (the right hand side of the colon). This local name can be used in subsequent statements within the unit. Furthermore, all of the exports (`serveWeb`) depend on all of the imports (`serveFile` and `serveCGI`). The unit’s implementation is in the file `web.c`, in Figure 6. The **rename** declarations resolve the conflict between importing and export-


```

bundletype Serve = { serve_web }
bundletype Stdio = { fopen, fprintf }
flags CFlags = { "-Ioskit/include" }

unit Web = {
  imports [ serveFile : Serve,
            serveCGI : Serve ];
  exports [ serveWeb : Serve ];
  depends {
    serveWeb needs (serveFile + serveCGI);
  };
  files { "web.c" } with flags CFlags;
  rename {
    serveFile.serve_web to serve_file;
    serveCGI.serve_web to serve_cgi;
  };
}

unit Log = {
  imports [ serveWeb : Serve,
            stdio : Stdio ];
  exports [ serveLog : Serve ];
  initializer open_log for serveLog;
  finalizer close_log for serveLog;
  depends {
    (open_log + close_log) needs stdio;
    serveLog needs (serveWeb + stdio);
  };
  files { "log.c" } with flags CFlags;
  rename {
    serveWeb.serve_web to serve_unlogged;
    serveLog.serve_web to serve_logged;
  };
}

unit LogServe = {
  imports [ serveFile : Serve,
            serveCGI : Serve,
            stdio : Stdio ];
  exports [ serveLog : Serve ];
  link {
    [serveWeb] <- Web <- [serveFile,serveCGI];
    [serveLog] <- Log <- [serveWeb,stdio];
  };
}

```

Figure 5: Unit descriptions for parts of a Web server

ing three functions with the same name by mapping the imported `serve_web` identifiers in bundles `serveFile` and `serveCGI` onto the C identifiers `serve_file` and `serve_cgi`.

The Log unit imports a `serve_web` function plus a bundle of I/O functions, and exports a `serve_web` function. The initialization and finalization declarations provide the same information as the graphical version of the

```

web.c:
err_t serve_web(socket_t s, char *path) {
  if (!strcmp(path, "/cgi-bin/", 9))
    return serve_cgi(s, path + 9);
  else
    return serve_file(s, path);
}

log.c:
static FILE *log;

void open_log() {
  log = fopen("ServerLog", "a");
}

err_t serve_logged(socket_t s, char *path) {
  int r;
  r = serve_unlogged(s, path);
  fprintf(log, "%s -> %d\n", path, r);
  return r;
}

```

Figure 6: Unit internals for parts of a Web server

unit in Figure 4. The dependency declarations specify that the functions `open_log` and `close_log` call functions in the `stdio` bundle, and all of the exports depend on all of the imports. Finally, the **rename** declarations resolve the conflict between the imported and exported `serve_web` functions, this time by renaming both the imported and exported versions.

The LogServe compound unit links the Web and Log units together, propagating imports and exports as in Figure 3. Specifically, in the **link** section,

```
[serveWeb] <- Web <- [serveFile,serveCGI]
```

instantiates a Web server unit using the `serveFile` and `serveCGI` imports, and binds the Web unit's exported bundle to the local name `serveWeb`. The next line specifies that `serveWeb` is an import, along with `stdio`, when instantiating the Log unit. The exported bundle of the Log unit is bound to `serveLog`, which is also used in the **exports** declaration of the compound unit, indicating that Log's exports are propagated as exports from the compound unit.

Figure 6 shows the C code implementing the Web and Log units. Only a few details have been omitted, such as the `#include` lines. Within `web.c`, the names `serve_cgi` and `serve_file` refer to imported functions, and `serve_web` is the exported function. Similarly, in `log.c`, `serve_unlogged`, `fopen`, and `fprintf` are all imports, while `serve_logged` is an export and `open_log` is the initializer.

4 Checking Architectural Constraints

Beyond making components easier to describe and link, Knit is designed to enable powerful analysis and optimization tools for componentized systems code. In this sense, Knit serves as a bridge between low-level implementation techniques and high-level program analysis techniques.

Component kits, especially, need analysis tools to help ensure that components are assembled correctly—much more than libraries and fixed architectures need such tools. In the case of a fixed (but extensible) architecture, a programmer can learn to code by a certain set of rules and to debug each extension until it seems to follow the rules. In the case of a component kit as flexible as the OSKit, however, the rules of proper construction change depending on which components are linked together. For example, in a kernel using a “null implementation” of threads, components need not provide re-entrant procedures because the “null implementation” keeps all execution single-threaded. But when the thread component is replaced with an actual implementation of threads, the rules for proper construction of the system suddenly require re-entrant procedures.

These kinds of problems fall outside the scope of conventional checking tools such as static type systems. Type systems in most programming languages (including C, C++, etc.) express concrete properties of code, such as data representation and function calling conventions, and do not express abstract properties like deadlock avoidance or whether code is in the top half or bottom half of a device driver. More importantly, conventional type systems detect *local errors*, but the problems that occur in component software are often *global errors*, where each individual component composition may be correct, but the entire system is wrong.

To start exploring the space of possible analyses over component-based programs, we have included in Knit a simple, extensible constraint system. This system allows a programmer to define properties that Knit should check, and then lets the programmer annotate each unit declaration with the properties it satisfies.⁴

As an example property, consider the distinction between “top half” code, which includes functions like `pthread_lock` or `sleep` that require a process context, and “bottom half” code, which includes interrupt handlers that work without a context. We would like Knit to enforce the constraint that bottom-half code does not directly call top-half code, an error that might happen when a set of components is wired together incorrectly.

⁴Besides their value as checkable properties, constraints provide useful documentation of the component’s behavior. Indeed, in our experience so far, constraints often duplicate information provided informally in documentation.

We can define this property in Knit with the following declarations:

```
property context
type NoContext
type ProcessContext < NoContext
```

which declare a property `context` and its two possible values, with a partial ordering on the property values that indicates `NoContext` is more general than `ProcessContext`.

Given this definition, a programmer can annotate imports and exports in units to establish property constraints. The examples below illustrate the three most common forms of annotation:

```
context(pthread_lock) <= ProcessContext
context(panic)         >= NoContext
context(sprintf)        <= context(putchar)
```

These three forms of constraint indicate that (1) a function (in this case, `pthread_lock`) requires a process context; (2) a function (e.g., `panic`) must work in situations where there is no process context; and (3) a function (e.g., `printf`) cannot be more flexible than some other function (e.g., `putchar`, which is used to implement `printf`). Note that the last form of constraint allows the constraints of one component to be propagated through other components in a chain of links. In practice, we find that such propagation of constraints appears most often, since most components are flexible enough to adapt to many constraint environments.

When components are linked together, Knit analyzes the components’ constraints and reports an error if the constraints cannot be satisfied for some property (or if an expected constraint declaration is missing). When Knit reports a property failure, it displays the shortest chain of constraints that demonstrates the source of the problem.

5 Experience

So far, we have applied Knit to two different sets of components: (1) the OSKit [10], a large set of components that includes many legacy components, and (2) a partial implementation of the Click modular router [25], comprising a few new and cleanly constructed components.

As reported in the following sections, our experience has been positive, but with two caveats. First, the current implementation of Knit is a prototype, and the only users to date are its implementors. Second, even the implementors are unsatisfied with the current Knit syntax, which leads to linking specifications that seem excessively verbose for many tasks.

5.1 Knit and the OSKit

The OSKit is a collection of components for building operating systems. Rather than defining a fixed structure

for an operating system, the OSKit provides raw materials for implementing whatever system structure a user has in mind. OSKit components can be combined in endless ways, and users are expected to write their own extensions and replacements for many kinds of components, depending on the needs of their designs. The components range in size from large, such as a TCP/IP stack derived from FreeBSD (over 18000 lines of non-blank/non-comment/etc. code), to small, such as serial console support (less than 200 lines of code). To use these components, OSKit users—many of whom know little about operating systems—must understand the interface of each component, including its functional dependencies and its initialization dependencies.

Before Knit: In the initial version of the OSKit, each component was implemented by one or more object (.o) files, which were stored in library archives, linked via `ld`. A component could be replaced by providing a replacement object file/library before the original library in the `ld` linking line. Since `ld` inspects its arguments in order, and since it ignores archive members that do not contribute new symbols (referenced by previously used objects), a careful ordering of `ld`'s arguments would allow a programmer to override an existing component.

As the OSKit grew in size and user base, experience soon revealed the deficiencies of `ld` as a component-linking tool. As depicted in Figure 1(c), interposition on component interfaces was difficult. Similarly, components that provided different implementations of the same interface would clash in the global namespace used for linking by `ld`. Even just checking that the linked set of components matched the intended set was difficult.

To address these issues (and, orthogonally, to represent run-time objects such as open files), a second version of the OSKit introduced COM abstractions for many kinds of components. For example, the system console, thread blocking, memory allocation, and interrupt handling are all implemented by COM components in the OSKit. For convenience, these COM objects are typically stored in a central “registry component.”

Although adding COM interfaces to the OSKit solved many of the technical issues with `ld` linking, in some ways it worsened the usability problems. Programmers who had successfully used the simple function interfaces in the original OSKit at first rebelled at having to set up seemingly gratuitous objects and indirections. Programmers became responsible for getting reference counts right and for linking objects together by explicitly passing pointers among COM instances. In practice, merely getting the reference counting right was a significant barrier to experimenting with new system configurations. Furthermore, inconvenient COM interfaces proved contagious. For example, to support a kernel in which different parts of the system use different memory pools,

the memory allocator component had to be made a COM object. This required changes to all code that uses allocators, changes to the code that inserts objects into the registry, and careful tweaking of the initialization order to try to ensure that objects in the registry were allocated with and subsequently used the correct allocators.

With both `ld` and COM, component linking problems interfere with the main purpose of the OSKit, which is to be a vehicle for quick experimentation. The motivation for Knit is to eliminate these problems, allowing programmers to specify which components to link together as directly as possible.

After Knit: We have converted approximately 250 components—about half of the OSKit—and about 20 example kernels to Knit. The process of developing Knit declarations for OSKit components revealed many properties and interactions among the components that a programmer would not have been able to learn from the documentation alone. Annotating a component took anywhere from 15 minutes (typically) to a full day (rarely), depending partly on the complexity of the component and its initialization requirements but mostly on the quality of the documentation (e.g., whether the imports and exports were clear).

Using Knit, we can now easily build systems that we could not build before without undue effort. For example, OSKit device drivers generate output by calling `printf`, which is also used for application output. Redirecting device driver output without Knit requires creating two separate copies of `printf`, then renaming `printf` calls in the device drivers either through cut-and-paste (a maintenance problem) or preprocessor magic (a delicate operation). Interposing on functions requires similar tricks. Such low-tech solutions work well enough for infrequent operations on a small set of names, but they do not scale to component environments in which configuration changes are frequent. Using Knit, interposition and configuration changes can be implemented and tested in just a few minutes.

Knit's automatic scheduling of initialization code was a significant aid in exploring kernel configurations. In a monolithic or fixed-framework kernel, an expert programmer can write a carefully devised function that calls all initializers in the right order, once and for all. This is not an option in the OSKit, where the correct order depends on which components are glued together. Previous versions of the OSKit provided canned initialization sequences, but, as just described, using these sequences would limit the programmer's control over the components used in the configuration. Knit allows the expert to annotate components with their dependencies and allows client programmers to combine precisely the components they want with reliable initialization. Annotations for device drivers, filesystems, networking, con-

sole, and other intertwined components have proven relatively easy to get right at the local level, and the scheduler has performed remarkably well in practice.

The constraint system described in Section 4 caught a few small errors in existing OSKit kernels, written by ourselves, OSKit experts. We added constraints to kernels composed of roughly 100 units. Among those units, 35 required the addition of constraints, of which 70% simply propagated their context from imports to exports using the constraint “context(exports) <= context(imports)” or stated that a component could be used without a process context. These required little effort. The remainder (device drivers and thread packages) required more care because we had to examine the source code to determine how individual components were used. The errors we found were easy to fix once identified. The advantage of Knit is that its constraint system found the bugs, and will continue to detect new bugs as the code evolves.

A further benefit of using Knit is that it makes it easier to create small, special-purpose kernels. The combination of knowing exactly which components are in our kernels (and why) and the ease of replacing one component with another enabled us to dramatically reduce the size of some kernels. An extreme example is our smallest kernel (the toy `hello_world` kernel) which is four times smaller when built with Knit than without.

The Knit version of the OSKit continues to use COM for subsystems that behave more like objects than modules. For example, individual files and directories are still implemented as COM objects.

5.2 Clack

The elegant Click modular router [25] allows a programmer, or even a network administrator, to build a special-purpose router by wiring together a set of components. Click provides its own language for configuring routers, so that a programmer might write

```
FromDevice(eth) -> Counter -> Discard
```

to create a “router” that counts packets.

Click is implemented in C++, and each router component is implemented by a C++ class instance. A programmer can add new kinds of router components to Click by deriving new C++ classes. To demonstrate that Knit is general and more than just a tool for the OSKit, we implemented a subset of Click version 1.0.1 with Knit components instead of C++ classes.⁵ We dubbed our new component suite *Clack*.

⁵Because Click’s router components are generally very small and functionally simple, much of the actual component source code deals with the Click-specific component framework and not with the functional purpose of the components. For this reason, we decided to write our components from scratch rather than adapt the existing Click components to Knit.

Given Click as a model, implementing enough of Clack in Knit to build an IP router (without handling fragmentation or IP options) took a few days. A typical Clack component required several lines of C plus several lines of unit description. Clack follows the basic architecture of Click, but the details have been Knitified. For example, Click supports component initialization through user-provided strings. Clack emulates this feature with trivial components that provide initialization data. Similarly, Click’s support for a (configure-time) variable number of imports or exports is handled in Clack with appropriate fan-in and fan-out components. Clack does not emulate the more dynamic aspects of Click, such as allowing a component to locate certain other components at run time.

Overall, by avoiding the syntactic overhead required to retrofit C++ classes as components, Clack definitions are considerably more compact than corresponding Click definitions (by roughly a factor of three for small components). The size of Clack .o files was smaller than Click .o’s by an even more dramatic amount (roughly a factor of seven for small components). This is mainly due to Clack’s fine-grained control of the router’s content and Click’s support for dynamic composition. The overall performance of Clack is comparable to that of Click.

In contrast, using the full Knit linking language to join Clack components is more complex than using Click’s special-purpose language. If Clack were to be used by network administrators, we would certainly build a (straightforward) translator from Click linking specifications to Knit linking expressions.

Based on our small experiment, we believe that Knit would have been a useful tool for implementing the original Click component set. The Click architecture fits well in the Knit language model, and the Click configuration language is conceptually close to the Knit linking model. The one aspect of Click that does not fit well into Knit is the rapid deployment of new configurations. Click configurations consist of C++ object graphs that can be dynamically generated, whereas Clack configurations are resolved at link time. Note, however, that recent work on Click performance by its authors also conflicts with dynamic configuration [19].

To the extent that Knit is a bridge to analyses and optimizations, we believe that Knit would be a superior implementation environment for Click compared to C++. In Section 6, we report on cross-component optimizations in Knit, and we show that they substantially increase the performance of Clack. The constraint-checking facilities of Knit can also be used to enforce configuration restrictions among Clack components, ensuring, for example, that components only receive packets of an appropriate type (Ethernet, IP, TCP, ARP, etc.).

These analyses are only a start, and a Knit-based Click would be able to exploit future Knit developments.

6 Implementation and Performance

Component software tends to have worse performance than monolithic software. Introducing component boundaries invariably increases the number of function calls in a program and hides opportunities for optimization. However, Knit's static linking language allows it to eliminate these costs. Indeed, we can achieve useful levels of optimization while exploiting the existing infrastructure of compilers and linkers.

In a typical use, the Knit compiler reads the linking specification and unit files, generates initialization and finalization code, runs the C compiler or assembler when necessary, and ultimately produces object files. The object files are then processed by a slightly modified version of GNU's `objcopy`, which handles renaming symbols and duplicating object code for multiply-instantiated units. Finally, these object files are linked together using `ld` to produce the program.

To verify that Knit does not impose an unacceptable overhead on programs, we timed Knit-based OSKit programs that were designed to spend most of their time traversing unit boundaries. We compared these programs with equivalent OSKit programs built using traditional tools. The number of units in the critical path ranged between 3 and 8 (including units such as memory file systems, VGA device drivers, and memory allocators), with the total number of units between 37 and 72. Tests were run between 10 and a million times, as appropriate. Knit was from 2% slower to 3% faster, $\pm 0.25\%$. Note that these experiments were done without applying the optimization that we describe next.

For cross-component optimization, we have implemented a strategy that is deceptively simple to describe: Knit merges the code from many different C files into a single file, and then invokes the C compiler on the resulting file. The task of merging C code is simple but tedious; Knit must rename variables to eliminate conflicts, eliminate duplicate declarations for variables and types, and sort function definitions so that the definition of each function comes before as many uses as possible (to encourage inlining in the C compiler). Fortunately, these complexities are minor compared to building an optimizing compiler. To limit the size of the file provided to the compiler, Knit can merge files at any unit boundary, as directed by the programmer via the unit specifications. When used in conjunction with the GNU C compiler (which has poor interprocedural optimization), this enables functions to be inlined across component boundaries which may, in turn, enable further interprocedural optimizations such as constant folding and

common subexpression elimination.⁶

To test the effectiveness of Knit's optimization technique (which we call *flattening*), we applied it to our Clack IP router. Since our focus was on the structure of the router, we flattened only the router rather than the entire kernel. For comparison, we rewrote our router components in a less modular way: combining 24 separate components into just 2 components, converting the result to idiomatic C, and eliminating redundant data fetches. The most important measure of an optimization is, of course, the time the optimized program takes. In this case, we also measured the impact of stalls in the instruction fetch unit because there is a risk that the inlining enabled by flattening would increase the size of the router code, leading to poor I-cache performance.⁷ Our experiments were performed on three 200MHz Pentium Pro machines, each with 64 MB of RAM and 256 KB of L2 cache, directly connected via DEC Tulip 10/100 Ethernet cards, with the "machine in the middle" functioning as the IP router.

The results are shown in Table 1. The manual transformation gives a significant (21%) performance improvement, demonstrating that componentization can have significant overhead. Flattening the modular version of the router gives an even more significant (35%) improvement: rather than harming I-cache behavior, flattening greatly improves I-cache behavior. Examination of the assembly code reveals that flattening eliminates function call overhead (e.g., the cost of pushing arguments onto the stack), turns function call nests into compact straight-line code, and eliminates redundant reads via common subexpression elimination. Combining both optimizations gives only a small (5%) additional improvement in performance, suggesting that the optimizations obtain their gains from the same source. Our overall conclusion is that we can eliminate most of the cost of componentization by blindly merging code, enabling conventional optimizing compilers to do the rest.

Meanwhile, the authors of Click have been working on special-purpose optimizations for their system [19]. Their optimizations include a "fast classifier" that generates specialized versions of generic components, a "specializer" that makes indirect function calls direct, and an "xform" step that recognizes certain patterns of components and replaces them with faster ones. While their code base and optimizations are very different from ours, the relative performance of their system and the effectiveness of their optimizations provides a convenient touchstone for our results. The performance of their base

⁶We used gcc version 2.95.2 for all our experiments.

⁷We also measured number of instruction misses in the L1 and L2 caches: both the overall downward trend and the approximate ratio between the three numbers were the same across all experiments.

hand optimized	flattened	cycles	instr. fetch stall cycles	text size (bytes)
✓	✓	2411	781	109464
		1897	637	108246
✓	✓	1574	455	106065
		1457	361	106305

Table 1: Clack router performance using various optimizations, measured in number of cycles from the moment a packet enters the router graph to the moment it leaves. I-fetch stalls were measured using the Pentium Pro counters and are reported in cycles. The i-fetch stall numbers along with the given code sizes reveal that inlining did not have a negative effect.

version	cycles
unoptimized	2486
optimized	1146

Table 2: Click router performance, with and without all three MIT optimizations, measured as above. The Click routers were executed in the same OSKit-derived kernel and on the same hardware as the Clack routers.

and optimized systems is shown in Table 2. We note that the performance of their base system is approximately the same as ours (3% slower) but that the effect of applying all three Click optimizations is significantly better than the two Clack optimizations (54%). Considering that Knit achieves its performance increase by blindly merging code, without any profiling or tuning of Clack by programmers, we again interpret the results of our experiment to indicate that Knit would make a good implementation platform for Click-like systems. We believe that Knit would save implementors of such systems time and energy implementing basic optimizations, allowing them to concentrate on implementing domain-specific or application-specific optimizations.

The core of our current Knit compiler prototype consists of about 6000 lines of Haskell code, of which roughly 500 lines implement initializers and finalizers, 500 lines implement constraints, and 1500 lines implement flattening. Our prototype implementation is acceptably fast—more than 95% of build time is spent in the C compiler and linker—although constraint-checking more than doubles the time taken to run Knit.

7 Related Work

Much of the early research in component-based systems software involves the design and implementation of microkernels such as Mach [1] and Spring [13]. With an emphasis on robustness and architectures for flexibility and extensibility at subsystem boundaries, microker-

nel research is essentially complementary to research on component implementation and composition tools like Knit. More recently, the Pebble [11] microkernel-based OS has an emphasis on flexibly combining components in protection domains, e.g., in separate servers or in a single protection domain. However, the actual set of components is quite fixed.

More closely related research involves the use of component kits for building systems software. MMLite [16] followed our OSKit lead by providing a variety of COM-based components for building low-level systems. MMLite takes a very aggressive approach to componentization and provides certain features that the current OSKit and Knit lack, such as the ability to replace system components at run-time. The Scout operating system [24] and its antecedent x-kernel [17] consist of a modest number of modules that can be combined to create software for “network appliances” and protocols. The Click system [25] also focuses on networking, but specifically targets packet routers (e.g., IP routers) and its components are much smaller than Scout’s. Scout and Click, like Knit, rely on “little languages” outside of C to build and optimize component compositions, and to schedule component initializations, but only Knit provides a general-purpose language that can be used to describe both new and existing components. Languages like C++ and Java have also dealt with automatic initialization of static variables, but through complex (and, in the case of C++, unpredictable) rules that give the programmer little control.

Knit’s ability to work with unmodified C code distinguishes it from projects such as Fox [15] and Ensemble [20], which rely on a high-level implementation language, or systems such as pSOS [31] and the currently very small eCos [29]. Both eCos and pSOS provide configuration languages/interfaces but neither really has “components”: individual subsystems can be included or excluded from the system, but there is no way to change the interconnections between components. In contrast to Ensemble, it should be noted that Knit provides a more “lightweight” system for reasoning about component compositions. This is a deliberate choice: we intend for Knit to be usable by systems programmers without training in formal methods.

Like Knit, OMOS [27,28] enables the reuse of existing code through interface conversion on object files (e.g., renaming a symbol, or wrapping a set of functions). Unlike Knit, however, OMOS does not provide a configuration language that is conducive to static analysis.

Commercial tools, such as Visual Basic and tools using it, help programmers design, browse, and link software components that conform to the COM or CORBA standards. Such tools and frameworks currently lack the kinds of specification information that would en-

able automated checking of component configurations beyond datatype interfaces, and the underlying object-based model of components makes cross-component optimization exceedingly difficult.

Knit's unit language is derived from the component model of Flatt and Felleisen [9], who provide an extensive overview of related programming languages research. Module work that subsequently achieved similar goals includes the recursive functors of Crary et al. [5] and the typed-assembly linker of Glew and Morrisett [12].

CM [3] solves for ML many of the same problems that Knit solves for C, but ML provides CM with a pre-existing module language and a core language with well-defined semantics. Unlike Knit, CM disallows recursive modules, thus sidestepping initialization issues. Further, CM relies on ML's type system to perform consistency checks, instead of providing its own constraint system.

The lazy functional programming community routinely uses higher-order functions to glue small components into larger components, relies on sophisticated type systems to detect errors in component composition, and makes use of lazy evaluation to dynamically determine initialization order in cyclic component graphs. For example, the Fudgets GUI component library [4] uses a dataflow model similar to the one used in Click (and Clack) but the data sent between elements can have a variety of types (e.g., menu selections, button clicks, etc.) instead of a single type (e.g., packets). This approach has been refined and applied to different domains by a variety of authors.

The GenVoca model of software components [2] has several similarities to our work: their "realms" correspond to our bundle types, their "type equations" correspond to our linking graphs, and their "design rules" correspond to our constraint systems. In its details, however, the GenVoca approach is quite different from ours. GenVoca is based on the notion of a *generator*, which is a compiler for a domain-specific language. GenVoca components are program transformations rather than containers of code; a GenVoca compiler therefore synthesizes code from a high-level (and domain-specific) program description. In contrast, Knit promotes the reuse of existing (C) code and enables flexible composition through its separate, unit-based linking language. Notably, Knit allows cyclic component connections—important in many systems—and can check constraints in such graphs. The GenVoca model of components and design rules, however, is based on (non-cyclic) component trees.

8 Conclusion

From our experiences in building and using OSKit components, and that of our clients in using them, we believe

that existing tools do not adequately address the needs of componentized software. To fill the gap, we have developed Knit, a language for defining and linking systems components. Our initial experiments in applying Knit to the OSKit show that Knit provides improved support for component programming.

The Knit language continues to evolve, and future work will focus on making components and linking specifications easier to define. In particular, we plan to generalize the constraint-checking mechanism to reduce repetition between different constraints and, we hope, to unify scheduling of initializers with constraint checking. We may also explore support for dynamic linking, where the main challenge involves the handling of constraint specifications at dynamic boundaries. Continued exploration of Knit within the OSKit will likely produce improvements to the language and increase our understanding of how systems components should be structured.

Knit is a first step in a larger research program to bring strong analysis and optimization techniques to bear on componentized systems software. We expect such tools to help detect deadlocks, detect unsafe locking, reduce abstraction overheads, flatten layered implementations, and more. All of these tasks require the well-defined component boundaries and static linking information provided by Knit. We believe that other researchers and programmers who are working on componentized systems could similarly benefit by using Knit.

Availability

Source and documentation for our Knit prototype is available under <http://www.cs.utah.edu/flux/>.

Acknowledgments

We are grateful to Mike Hibler, Patrick Tullmann, John Regehr, Robert Grimm, and David Andersen for providing many comments and suggestions that helped us to improve this paper. Mike deserves special thanks, for special help. We are indebted to the members of the MIT Click group, both for Click itself and for sharing their work on Click optimization long before that work was published. Finally, we thank the anonymous reviewers and our shepherd, David Presotto, for their constructive comments and suggestions.

References

- [1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proc. of the Summer 1986 USENIX Conf.*, pages 93–112, June 1986.
- [2] D. Batory and B. J. Geraci. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering*, pages 67–82, Feb. 1997.

- [3] M. Blume and A. W. Appel. Hierarchical Modularity. *ACM Transactions on Programming Languages and Systems*, pages 812–846, July 1999.
- [4] M. Carlsson and T. Hallgren. Fudgets: A Graphical User Interface in a Lazy Functional Language. In *Proc. of the Conference on Functional Programming and Computer Architecture*, 1993.
- [5] K. Crary, R. Harper, and S. Puri. What is a Recursive Module? In *Proc. ACM SIGPLAN '99 Conf. on Programming Language Design and Implementation (PLDI)*, pages 50–63, Atlanta, GA, May 1999.
- [6] S. Dorward and R. Pike. Programming in Limbo. In *Proc. of the IEEE Compcon 97 Conf.*, pages 245–250, San Jose, CA, 1997.
- [7] R. B. Findler and M. Flatt. Modular Object-Oriented Programming with Units and Mixins. In *Proc. of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, pages 94–104, Baltimore, MD, Sept. 1998.
- [8] M. Flatt. *Programming Languages for Component Software*. PhD thesis, Rice University, June 1999.
- [9] M. Flatt and M. Felleisen. Units: Cool Units for HOT Languages. In *Proc. ACM SIGPLAN '98 Conf. on Programming Language Design and Implementation (PLDI)*, pages 236–248, June 1998.
- [10] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A Substrate for OS and Language Research. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, pages 38–51, St. Malo, France, Oct. 1997.
- [11] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The Pebble Component-Based Operating System. In *Proc. of the USENIX 1999 Annual Technical Conf.*, June 1999.
- [12] N. Glew and G. Morrisett. Type-Safe Linking and Modular Assembly Language. In *Proc. of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 250–261, San Antonio, TX, Jan. 1999.
- [13] G. Hamilton and P. Kougouris. The Spring Nucleus: a Microkernel for Objects. In *Proc. of the Summer 1993 USENIX Conf.*, pages 147–159, Cincinnati, OH, June 1993.
- [14] S. P. Harbison. *Modula-3*. Prentice Hall, 1991.
- [15] B. Harper, E. Cooper, and P. Lee. The Fox Project: Advanced Development of Systems Software. Computer Science Department Technical Report 91–187, Carnegie Mellon University, 1991.
- [16] J. Helander and A. Forin. MMLite: A Highly Componentized System Architecture. In *Proc. of the Eighth ACM SIGOPS European Workshop*, pages 96–103, Sintra, Portugal, Sept. 1998.
- [17] N. C. Hutchinson and L. L. Peterson. Design of the x-kernel. In *Proc. of SIGCOMM '88*, pages 65–75, Aug. 1988.
- [18] International Organization for Standardization. *Ada 95 Reference Manual. The Language. The Standard Libraries*, Jan. 1995.
- [19] E. Kohler, B. Chen, M. F. Kaashock, R. Morris, and M. Poletto. Programming language techniques for modular router configurations. Technical Report MIT-LCS-TR-812, MIT Laboratory for Computer Science, Aug. 2000.
- [20] X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, and R. Constable. Building reliable, high-performance communication systems from components. In *Proc. of the 17th ACM Symposium on Operating Systems Principles*, pages 80–92, Dec. 1999.
- [21] D. MacQueen. Modules for Standard ML. In *Proc. of the 1984 ACM Conf. on Lisp and Functional Programming*, pages 198–207, 1984.
- [22] Microsoft Corporation and Digital Equipment Corporation. *Component Object Model Specification*, Oct. 1995. 274 pages.
- [23] J. G. Mitchell, W. Mayberry, and R. Sweet. *Mesa Language Manual*, 1979.
- [24] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, T. A. Proebsting, and J. H. Hartman. Scout: A Communications-oriented Operating System. Technical Report 94–20, University of Arizona, Dept. of Computer Science, June 1994.
- [25] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashock. The Click Modular Router. In *Proc. of the 17th ACM Symposium on Operating Systems Principles*, pages 217–231, Dec. 1999.
- [26] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, June 1999. Revision 2.3. OMG document formal/98–12–01. Part of the CORBA 2.3 specification.
- [27] D. B. Orr, J. Bonn, J. Lepreau, and R. Mecklenburg. Fast and Flexible Shared Libraries. In *Proc. of the Summer 1993 USENIX Conf.*, pages 237–251, June 1993.
- [28] D. B. Orr, R. W. Mecklenburg, P. J. Hoogenboom, and J. Lepreau. Dynamic Program Monitoring and Transformation Using the OMOS Object Server. In D. Lilja and P. Bird, editors, *The Interaction of Compilation Technology and Computer Architecture*. Kluwer Academic Publishers, 1994.
- [29] Red Hat, Inc. eCos: Embedded Configurable Operating System, Version 1.3. <http://www.redhat.com/products/ecos/>.
- [30] C. A. Szyperski. Import Is Not Inheritance — Why We Need Both: Modules and Classes. In *ECOOP '92: European Conf. on Object-Oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 19–32. Springer-Verlag, July 1992.
- [31] WindRiver, Inc. pSOS. <http://www.windriver.com/>.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits:

- Free subscription to *login:*, the Association's magazine, published eight times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see <http://www.usenix.org/membership/specialdisc.html> for details.

Supporting Members of the USENIX Association:

Addison-Wesley	Macmillan Computer Publishing,	Sendmail, Inc.
Earthlink Network	USA	Smart Storage, Inc.
Edgix	Microsoft Research	Sun Microsystems, Inc.
Interhack Corporation	Motorola Australia Software Centre	Sybase, Inc.
Interliant	Nimrod AS	Syntax, Inc.
JSB Software Technologies	O'Reilly & Associates Inc.	UUNET Technologies, Inc.
Lucent Technologies	Performance Computing	Web Publishing, Inc.

Supporting Members of SAGE:

Collective Technologies	Macmillan Computer Publishing,	O'Reilly & Associates Inc.
Deer Run Associates	USA	Remedy Corporation
Electric Lightwave, Inc.	Mentor Graphics Corp.	RIPE NCC
ESM Services, Inc.	Microsoft Research	SysAdmin Magazine
GNAC, Inc.	Motorola Australia Software Centre	Taos: The Sys Admin Company
	New Riders Press	Unix Guru Universe

For more information about membership, conferences, or publications,
see <http://www.usenix.org/>

or contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA.
Phone: 510-528-8649. Fax: 510-548-5738. Email: office@usenix.org.

